Detecting Feature-Interaction Hotspots in Automotive Software using Relational Algebra

by

Bryan J. Muscedere

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

Waterloo, Ontario, Canada, 2018

Abstract

Modern software projects are programmed by multiple teams, consist of millions of lines of code, and are split into separate components that, during runtime, may not be contained in the same process. Due to these complexities, software defects are a common reality; defects cost the global economy over a trillion dollars each year. One area where developing safe software is crucial is the automotive domain. As the typical modern vehicle consists of over 100 million lines of code and is responsible for controlling vehicle motion through advanced driver-assistance systems (ADAS), there is a potential for these systems to malfunction in catastrophic ways.

Due to this risk, automotive software needs to be inspected to verify that it is safe. The problem is that it can be difficult to carry out this detection in code; manual analysis does not scale well, search tools like `grep` have no contextual awareness of code, and although code reviews can be effective, they cannot target the entire codebase properly. Furthermore, automotive systems are comprised of numerous, communicating features that can possibly interact in unexpected or undefined ways. This thesis addresses this problem through the development of a static-analysis methodology that detects custom interaction patterns coined as hotspots. We identify several classes of automotive hotspots that describe patterns in automotive software that have the possibility of manifesting as a feature interaction.

To detect these hotspots, this methodology employs a static, relational analysis toolchain that create a queryable model from source code and enable engineer defined queries to be run on the model that aim to reveal potential hotspots in the underlying source code. The purpose of this methodology is not to detect bugs with surety but work towards an analysis methodology that can scale to automotive software systems.

We test this hotspot detection methodology through a case study conducted on the Autonomoose autonomous driving platform. In it, we generate a model of the entire Autonomoose codebase and run relational algebra queries on the model. Each script in the case study detects a type of hotspot we identify in this thesis. The results of each query are presented.

Acknowledgements

There are numerous people I would like to acknowledge in this thesis that provided me with countless opportunities to excel during my time at the University of Waterloo. My thesis would certainly not be possible without the time and dedication that each of these individuals put in.

First I would like to thank Dr. Joseph D'Ambrosio and his team at General Motors for their assistance with my research. They provided a lot of suggestions and support that allowed me to focus my research to the automotive domain. Further, I was able to draw upon experience gained through an internship with Dr. D'Ambrosio's group at General Motors. This internship allowed me to collaborate with numerous automotive domain experts; I will value this experience throughout my computer science career.

I would also like to thank to the Waterloo Autonomous Vehicle Lab (WAVELab) for providing me with access to the Autonomoose source code. As this project relied on verifying the toolchain on automotive software, this project would not have been possible without access to Autonomoose. In addition, special thanks to research engineer Dr. Antkiewicz for meeting with me to describe the Autonomoose software and to examine some of the hotspot instances detected in the vehicle software to check their value.

I would like to thank Dr. Ian Davis for his support during my time at the University of Waterloo. He helped build many of the original tools used and provided assistance in using these tools. Additionally, he supported my work developing the ClangEx and Rex fact extractors by improving the Clang API and by tackling certain issues inherent with developing C and C++ extractors. Further, he provided numerous suggestions that allowed me to develop extremely polished hotspots and tools.

I would also like to thank Dr. Michael Godfrey for his help on this thesis. His work, insight, and advice was extremely valuable and I greatly appreciate the time he spent assisting me with this research. Further, Dr. Godfrey has a great deal of experience that I was able to draw upon from his own experiences from past research endeavors and from developing and using the relational algebra toolchain in a variety of different settings.

Finally, I would like to express my utmost gratitude to my supervisor Dr. Joanne Atlee for her support during my thesis and for being such a positive role model. I learned so much from her during my time at Waterloo and this thesis certainly would not be possible without her guidance and support. Dr. Atlee provided a lot of feedback that improved my writing and research ability. Additionally, under her supervision, I was able to partake in many different opportunities that allowed me to improve my research and abilities as a computer scientist.

Dedication

Dedicated to Mom, Dad, Danielle, Montana, and Mabel. I could not have done it without your love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developing a large-scale software system is complex. It can involve hundreds of people spread across multiple teams and requires the development of numerous software artifacts including models and source code. Additionally, software continues to increase in complexity; many modern systems consist of over a million source lines of code (MLoC) divided across numerous components. To illustrate, Figure 1.1 highlights the size of several common, large-scale software systems[1]. Each of these systems have codebases of over 2 million lines of code meaning that no single person or team can have a full mental model of the underlying software. Problems can arise when integrating components that are independently developed; these components have the potential of interacting in unexpected or undefined ways which can introduce bugs into the project. A 2017 report from the Australian software testing firm Tricentis found that the global economic cost of faulty software was 1.1 trillion US dollars in 2016 [13]. This amount currently compares to the gross domestic product of Mexico [14]. Worse, the economic impact from such bugs is expected to increase as society continues to become ever more technology-centric.

More worrying is that software is increasingly responsible for safety. The software that manages airplanes, power plants, and medical devices all have the potential of risking human lives if not developed properly. For instance, it was discovered that a tool used by the UK's National Health Service (NHS) that calculated patient risk for medication prescription, miscalculated 300,000 patients' risk of heart attacks exposing them to unnecessary or dangerous drugs [15]. Another case of faulty medical software took place in the 1980s involving the Therac-25 radiation therapy machine [16]. The software caused the release of lethal doses of radiation to seven patients resulting in their death.

[1]Data comes from [4], [5], [6], [7], [8], [9], [10], [11], and [12].

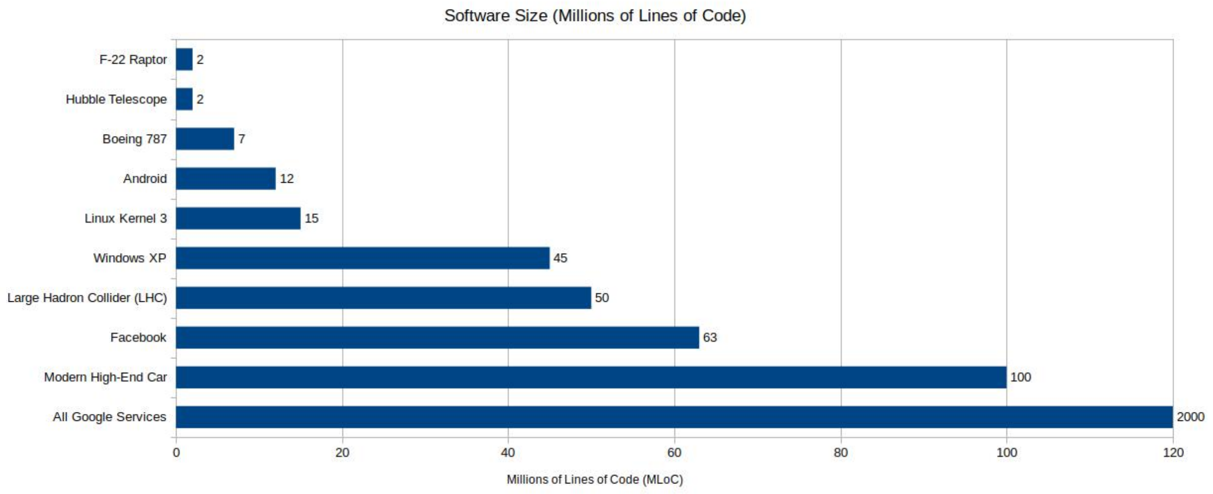


Figure 1.1: The number of lines of code for several software systems.

One safety-critical domain where software is increasingly playing a role is the automotive industry. According to Dr. Manfred Broy, the modern, high-end car contains over 100 million lines of code that operates across hundreds of electronic control units (ECUs) [4]. Although much of this code powers non-safety-critical systems such as infotainment, a lot of this software is now responsible for operating advanced driver assistance systems (ADAS). ADAS are specific features that are designed to help drivers operate the vehicle [17] and include features such as collision imminent braking (CIB) and adaptive cruise control (ACC). The CIB feature causes the car to brake or stop when a forward obstacle is detected, while ACC maintains a vehicle's speed based on a user set speed and the traffic around the vehicle. Although these systems are designed to improve driver safety, there have been numerous cases of these features operating improperly. For instance, in 2017, the Fiat-Chrysler automotive group recalled over 1.25 million trucks due to a software error that could cause side airbags and seat pretensioners to fail during a collision [18]. These recalls have safety and financial consequences; the failure of these airbags have been linked to one death in the United States.

At a high level, automotive systems comprise of numerous subsystems and features that communicate with each other. Although the definition of a feature varies depending on the granularity at which someone decides to work at [19], in this thesis we define a feature to be "a coherent and identifiable bundle of system functionality that helps characterize the functionality from a user perspective" [20]. If unchecked, ADAS features such as CIB and

ACC have the potential of interacting in an unexpected or unintended manner known as a *feature interaction*. Feature interactions amongst ADAS features can result in occupant injury or death. Figure 1.2 gives an example of a feature interaction between two ADAS features in the blue car marked with an "M": collision avoidance (CA) and side collision avoidance (SCA). Here, red car number one is merging onto the road causing the SCA feature in the blue car to move the car to the side to avoid a collision. Red car number two is accelerating from behind into the blue car causing the CA feature to accelerate the car forward to avoid a rear collision. This rapid change in forward and lateral acceleration could result in a vehicle rollover causing an accident.

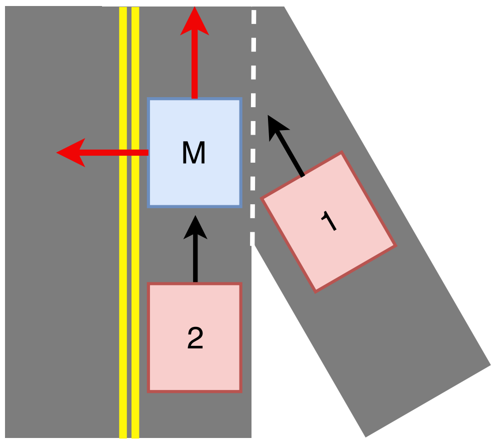


Figure 1.2: A case where vehicular features might interact in a vehicle (blue).

Detecting these feature interactions in code can be difficult due to codebase size and complexity. This problem is not new and researchers have developed numerous techniques to assist developers [19][21]. One popular technique is the use of manual code inspection to detect bugs and interactions. Although code reviews are an important quality assurance tool, they can be fairly time consuming. Effective code reviews should focus on only 200 to 400 lines of code at a time and should not exceed an inspection rate of 300 to 500 lines of code per hour [22]. Another detection method is the use of static analysis tools to detect problem areas and highlight calls between components [23]. Tools such as Understand or Klocwork are common and used to facilitate developer understanding of the source code and detect potential violations such as dereferencing NULL pointers or uninitialized variables. Although valuable, these type of tools do not have the capability of detecting high-level patterns such as feature interactions. Additionally, researchers have developed techniques to automatically detect feature interactions, but the techniques are compute intensive and do not scale to large software systems.

The aim of this thesis is to present a novel static-analysis toolchain to detect feature-interaction hotspots in the code of automotive software systems to avoid potential feature interactions. This toolchain creates a program model from software artifacts that can be queried using a context-aware language that allows for the detection of these hotspots. Because manual inspection is beneficial only on a small scale, the goal of this approach is to reduce the number of areas that need manual analysis by identifying these program hotspots. We argue that this approach complements current tools used in the automotive domain because of its flexibility, scalability, and ability to detect message-passing.

1.1 Hotspots

Because the precise approaches to feature interaction detection do not scale well to a large-scale system, other approaches must be used. These techniques do not detect feature interactions with surety but rather identify potential feature interactions that can then be inspected using other, more thorough, traditional analysis techniques.

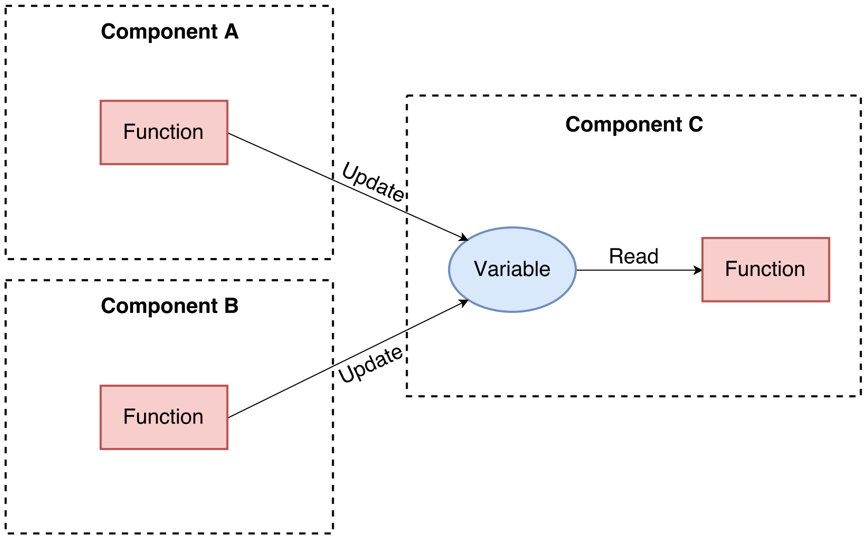


Figure 1.3: An example of a possible hotspot in a monolithic system.

A *hotspot* is a pattern that describes a possible feature interaction that can be detected using static analysis. Because feature interactions are domain dependent, the types of hotspots one might want to look for can vary greatly. For instance, a traditional software system with a single thread of control may not have to deal with inter-component communication or threads whereas an automotive system generally has to contend with significant communication among ECUs. For this analysis methodology to be effective, users of this toolchain need to express hotspots as patterns for the analysis to detect in the model.

Figure 1.3 shows a potential hotspot for a monolithic software system consisting of one thread of control and a single feature. In this example, there are three separate components that communicate with each other in some way. Components A and B both have a function that modifies Component C. Component C has a function that uses a variable. The value of this variable affects how the function in Component C operates. This type of situation could be deemed a hotspot because Components A and B potentially alter how Component C functions at some point in the program. If this hotspot instance is reported by the toolchain, system experts can then verify that the interactions between these three functions operate as expected.

Static analyses are evaluated with respect to their precision and recall of the target information [24][23]. Precision is defined as the proportion of true positives to the sum of true positives and false positives whereas recall is defined as the proportion of true positives to the sum of true positives and false negatives. An identified feature-interaction hotspot may point to a problematic area of code or may be a false positive. Precision and recall tend to be inversely related [25] and developing a "golden" static analysis tool that identifies hotspots with high precision and recall is impossible in practice. Due to this, it is important to develop techniques that have high recall because, with the automotive domain being safety critical, it is far more important to unnecessarily include and analyze any false positives than risk missing potential false negatives that could impact consumer safety. Further, false positives returned by our methodology can be filtered out using other static analysis methods or by manual analysis. Although it may seem tedious to pair our methodology with other static analysis tools, research by Willis et al. [26] found that analyzing source code with multiple static analysis tools allows for the more effective detection of issues in that project.

1.2 Thesis Overview

This section gives an overview of the two main steps of this methodology: the *generation of a program model* using fact extractors and the *generation of a hotspot report*. Detailed information describing the technology behind this methodology is described in Chapter 2.

Importantly, this methodology is generic and changes depending on the software project being examined, the domain of the project, and the hotspots being searched for. Subsequent sections of this thesis (Section 3 and 4) describe this process in more detail for automotive systems.

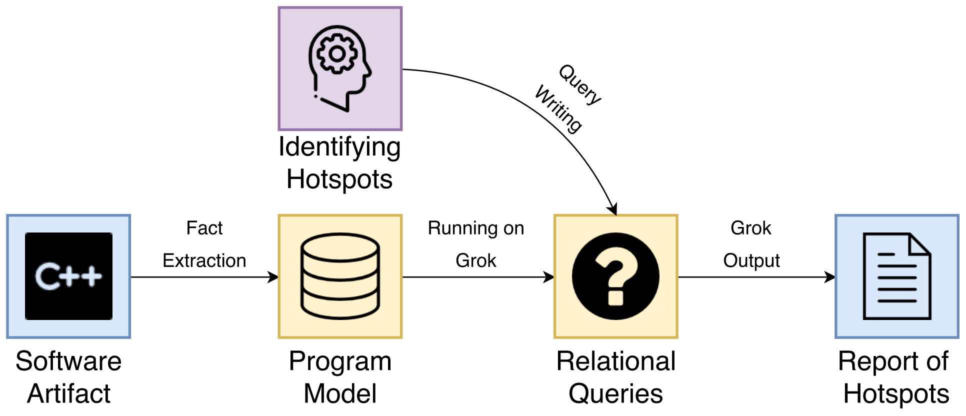


Figure 1.4: The hotspot detection toolchain.

1.2.1 Generation of Program Model

A high-level overview of our hotspot detection methodology is shown in Figure 1.4. In this figure, blue boxes are files, yellow boxes are intermediate files produced by the toolchain, and the single purple box represents a human-based thought process which identifies these hotspots.

Before hotspots can be detected in an automotive system, a model representing the underlying automotive software needs to be created. This model is generated statically from the source code and must contain enough information about the software that will allow for the identification of feature-interaction hotspots in subsequent steps. The information that needs to be included in these models to detect feature-interaction hotspots varies depending on the automotive software architecture and programming language. However, from a general perspective, information about features, how these features communicate, and how information flows between features should be recorded.

The generation of a program model occurs automatically by running artifacts from the software project through a custom, human-defined analyzer known as a *fact extractor*.

This step is crucial because it keeps only information about the software project that is essential for detecting specific hotspots. Additionally, fact extraction transforms the software project into a condensed model that is in a format that can easily be queried. As software projects vary in programming language, different fact extractors must exist for each language. Although it takes some effort to elicit fact extractor requirements and to determine the type of information to include in the model, once an extractor is developed it can be used on any project that uses the language it targets.

In this thesis, we developed two custom fact extractors called ClangEx and Rex that are designed to detect feature-interaction hotspots in automotive code. ClangEx is a general C and C++ fact extractor that serves as a basic fact extractor and collects program information like `variables`, `functions`, and relationships amongst these entities like function calls and inheritance hierarchies. Rex builds upon ClangEx by extracting information about C++ language features as well as components, message passing information, and feature communication in systems that use ROS for inter-component communications.

1.2.2 Generation of Hotspot Report

Given a model that describes the automotive software system, feature-interaction hotspots can be identified. First, before they can detected, these hotspots need to be characterized. This is a manual, thought-based task where developers of the system need to determine what they define as a hotspot.

In this thesis, we characterize three different types of automotive hotspots that might be present in such a system: *feature-communication*, *multiple-input*, and *control-flow* hotspots. Feature-communication hotspots look at how automotive features communicate in a system and which features they communicate with directly and indirectly. Multiple-input hotspots discover features that receive inputs from multiple features. Lastly, control-flow hotspots look for features that alter another feature's behaviour based on messages sent.

Once characterized, queries can be written that express specific patterns that the tools can use to comb through target software to look for those pattern matches. These queries are written in a specialized query language known as Grok [27] (see Section 2.2.3). The result of a query is a list of areas of code that match the hotspot pattern. We developed Grok scripts that detect each hotspot described in this thesis and ran them on the Autonomoose autonomous car project.

A third step that is technically part of this methodology, but not covered in this thesis, is the action that is taken after the hotspots are identified. These hotspot reports do not necessarily show areas of code that contain problems; rather, these reports should be used in tandem with manual code reviews or additional static or dynamic analysis tools to find bugs. These reports merely provide developers with a list of areas of code that need to be inspected more thoroughly.

1.3 Thesis Contributions

The contributions made in this thesis are as follows:

- The development of two different fact extractors that automatically produce program models from software artifacts. Each of these extractors collect information about the underlying source code with the goal of detecting hotspots. These fact extractors both generate models based on a particular schema. The two fact extractors are as follows:
  - **ClangEx**: A generic C and C++ fact extractor that is capable of extracting numerous language features from both languages. The purpose of ClangEx is to serve as a model fact extractor that can be easily extended to target more specific things in C or C++ projects. This extractor uses the Clang open-source compiler to operate and is able to process any source code that is adherent to ANSI C or ISO C++.
  - **Rex**: A C++ fact extractor based upon *ClangEx* that is capable of extracting messages passed between components. Rex looks for messages sent and received using the Robot Operating System (ROS) framework. It is the first extractor developed that captures message-passing in a distributed system.
- The identification of several different types of hotspots for automotive software systems. With automotive systems being distributed across ECUs, having a large codebase, and having active-safety requirements, hotspots in these systems tend to be on a feature-interaction level. This thesis describes the feature-interaction hotspots that might be present Additionally, we discuss the type of information that would need to be extracted that would allow for the detection of these hotspots.
- A case study that uses the relational analysis toolchain to detect hotspots in the Autonomoose autonomous car platform. The case study demonstrates the feasibility of detecting these hotspots in an automotive system that makes use of message-passing between components. Overall, it was found that the Rex extractor generated models with over 80% precision and recall with exception to one relation that had

78.75% recall. Additionally, a total of 1,460 instances of hotspots were found. Of these, 19% on average were deemed to be worthy of further inspection.

1.4 Thesis Organization

This thesis is organized into several sections. Chapter 2 gives a background on related work and and other static analysis tools. This chapter also provides a detailed description of the relational analysis toolchain and the ROS framework.

Chapter 3 gives a description of the two fact extractors that were developed during this thesis. These fact extractors are capable of extracting information from certain types of software artifacts and allows users to gain insight into how components in a software system interact. This chapter also compares the differences between these two extractors.

Chapter 4 describes several different types of hotspots that might be present in automotive systems and provides a brief description on how they can be detected using relational algebra.

Chapter 5 describes a major case study that was conducted that demonstrates the feasibility of detecting hotspots using the relational analysis toolchain on an automotive software system. This chapter describes the methodology of this case study, provides the results, and shows example relational algebra scripts that carry out this detection. The case studies in this section were conducted on the Autonomoose project.

Last, Chapter 6 provides a summary of the work completed in this thesis as well as limitations and future work that needs to be conducted.

Lastly, Appendix A is a user manual for the ClangEx and Rex extractors and Appendix B is a user manual for the bfx64 extractor. bfx64 is object-file extractor developed during our research but not covered in this thesis. These manuals provide installation instructions and usage details.

Chapter 2

Background

This chapter gives an overview of state-of-the-art static-analysis research and current industry analysis tools used in software development. Additionally, this chapter describes the relational-algebra toolchain that is used to detect hotspots in automotive systems. Understanding current research in academia and the tools used in industry is important to understand the strengths and weaknesses of current technology.

Section 2.1 highlights previous research in academia and industry. Section 2.2 provides an in-depth description of the relational-analysis toolchain used in our thesis to detect hotspots.

2.1 Related Work and Tools

Research into static-analysis techniques spans decades and is part of the software lifecycle for some organizations. Much modern static-analysis research aims to improve current methods that are used in statically analyzing a software project. This research works towards building techniques that are scalable, accurate, and informative. While research continues, there are already numerous static-analysis tools that are used in industry to detect defects in code and to assist developers with program comprehension.

This section will explore related areas of academic work and industrial practice. Section 2.1.1 looks at research that proposes different approaches to statically detect interactions that occur between subsystems in a software project. Section 2.1.2 looks at two related static-analysis tools that many development companies use to build software.

2.1.1 Related Research

There is much research that aims to develop approaches to statically detect interactions amongst components in distributed environments. This section explores two state-of-the-art techniques: **JITANA**, a technique to detect interactions amongst Android applications, and a method by Purandare et al. to analyze **distributed robotic systems**. Other areas of related work are briefly discussed.

JITANA

Tsutano et al. developed a novel analysis framework to detect interactions among applications that execute on the Android platform [1]. Known as JITANA, this framework operates by connecting to an Android device using the Android Debug Bridge (ADB) protocol and then extracting program information about selected Android programs. Information about these programs comes from the Java Class Loader and analysis engines that interface with JITANA's analysis controller. Once information is obtained from the ADB, JITANA is able to generate graphs showing component communication. Figure 2.1 shows a high-level overview of this architecture.

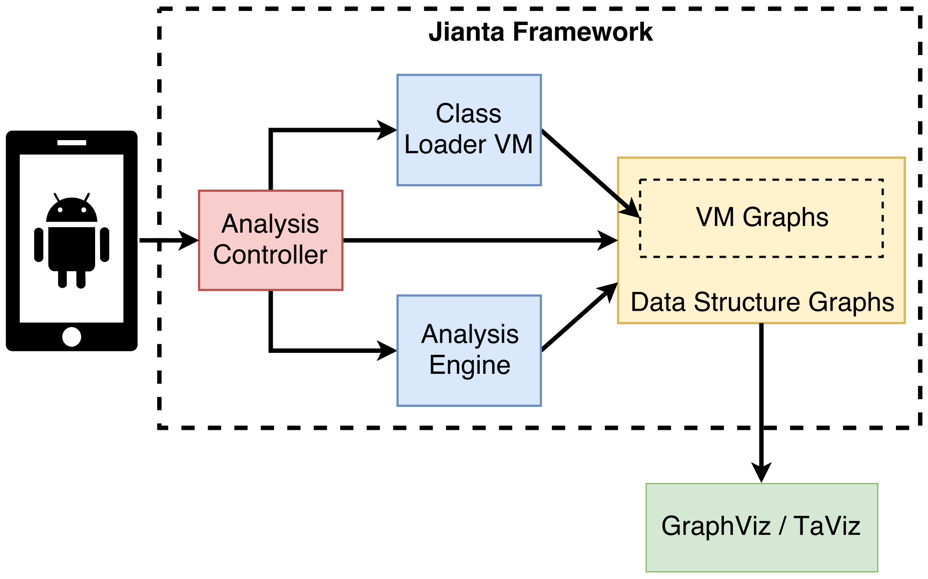


Figure 2.1: The JITANA analysis architecture. Adapted from [1].

Prior solutions for analyzing cross-application communications required combining separate Android packages (APKs) into a single APK and then viewing cross-application messages as inter-component communications (ICC). Although this methodology works on small Android applications, merging APKs into a single APK is a fragile process that fails on complex, commercial APKs. To avoid this, JITANA forgoes APK merging and leveraging the Android class loader by connecting directly to the phone. From it JITANA is able to generate class, method, and class loader graphs. Each of these graphs are represented as a hierarchical graph and can be converted to common graph formats like GraphViz or TraViz.

Although JITANA is an innovative way to analyze application interactions on the Android platform, it is not necessarily suited to detecting unwanted interactions amongst vehicular features. The reason it would not be suited for the automotive domain is because it captures all instances inter-application communications (IAC) between all the target applications being analyzed. For instance, they reported 63 interactions between 99 applications tested at the same time [1]. Although the authors do not state whether these 63 interactions are expected or undesired, this would be difficult to apply to the automotive domain where features communicate with each other as a normal part of vehicle operation. Likely, a user in this case would be inundated with interactions that are expected and be unable to discern undesired feature interactions from the report.

Distributed Robotic Analysis

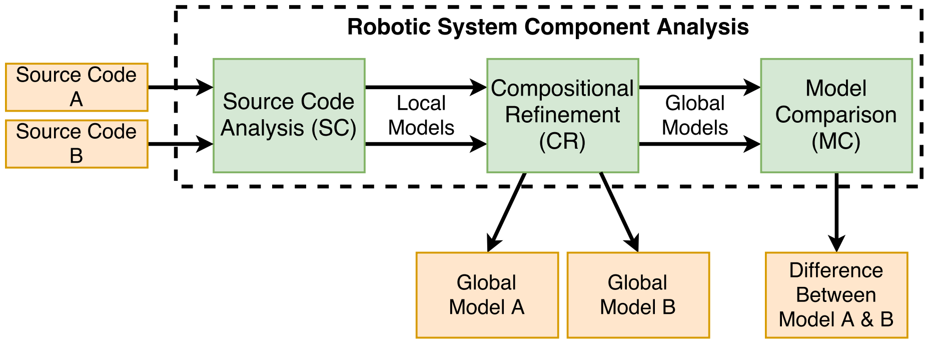

Robotic System Component Analysis
Source Code A
Source Code B
Source Code Analysis (SC)
Local Models
Compositional Refinement (CR)
Global Models
Model Comparison (MC)
Global Model A
Global Model B
Difference Between Model A & B

Figure 2.2: The robotic analysis architecture. Adapted from [2].

Purandare et al. developed an efficient method to extract conditional component dependencies from robotic systems [2]. Their analysis reports on interactions between components that arise due to program control-flow decisions. Specifically, this technique captures control-flow decisions and discovers dependencies between components and the conditions that cause these dependencies to occur. The result of this analysis is either a single model that highlights interactions that arise in some project or is a delta model from two software projects that shows interactions present in one but not the other. This delta model feature is useful as it can be used to see how code changes in a project affect interactions. For two multiset models generated by this approach, by checking $A \equiv B$ through $A \subseteq B$ and $A \supseteq B$, the model comparison unit shows only interactions that are unique in each model [2]. Figure 2.2 shows the architecture of this static-analysis technique.

Although this approach is capable of discovering messages between distributed components, this approach only shows components communicating and presents the conditions in which they communicate. As features are expected to communicate under normal conditions in an automotive system, the majority of the results would be expected interactions. This problem would become more prevalent as larger codebases are analyzed using this approach. As such, for this to be viable for detecting feature interactions, a mechanism to filter the results to discover problematic instances would need to be implemented. Further, although this approach looks at conditions where a message to another feature might occur, there might be other issues such as a feature having its behaviour changed as a result of a message. These types of interactions are not reported by this approach.

Other Work

In addition to work by Tsutano et al., and Purandare et al., there is other research on static analyses that aims to detect interactions between subsystems in event-based systems. For the purpose of this section, event-based systems are systems that produce, consume, and react to events sent between components [28].

Safi et al. [29] develop a static-analysis technique called DEvA to detect potential event anomalies. The authors define an event anomaly as a case where two or more different events cause accesses to the same field with at least one event causing a write [29]. DEvA operates by automatically extracting information about the event-based system and generating a report of event-based anomalies. DEvA was tested on 20 open-source systems and found event-based anomalies in each. Although these type of anomalies might be present in an automotive system if the definition of an event anomaly was adapted to fit automotive features, analyzing such systems using DEvA would not be sufficient because there might be other ways in which features interact besides “race-condition”-like situations.

Other researchers have attempted to use forms of algebra in defect detection and static analysis. Kozen [30] proposes using a form of relational algebra called Kleene Algebra with Tests (KAT) to assist with automatic verification tasks. KAT is an extension to Kleene Algebra that allows basic program constructs such as `while` loops to be modeled. Although Kozen demonstrates that KAT is effective in verifying such policies, manual effort is required to generate a KAT encoding of a code fragment and users need to inject assertions into the KAT encoding to test correctness. Although Kozen's approach is beneficial for verifying small, critical segments of code it would be difficult to apply to a large-scale automotive system.

2.1.2 Related Static-Analysis Tools

There are numerous static-analysis tools used in industry that assist developers with program comprehension and bug detection. Many of these tools tout similar features including simple syntactic code analysis, the generation of dependency diagrams, and the ability for developers to add their own syntactic code checks. Two notable static-analysis tools that are commonly used are **Understand** and **Klocwork**. The remainder of this section provides a quick overview of these two tools.

Understand

Understand is a commercial Integrated Development Environment (IDE) created by Scientific Toolworks that provides static-code analysis. The tool incorporates code-metric reporting and simple error detection, and can generate UML class and dependency diagrams of the underlying project [31]. Figure 2.3 shows the Understand tool visualizing the directory structure of the ClangEx fact-extractor project. In this figure, the user is presented with a dependency diagram that groups functions into the source files in which they are defined.

There are two major features in Understand that make it a valuable tool for detecting potential problem areas in source code. First, it has a feature called *CodeCheck* that checks the software project for a variety of different bugs. These bugs can be style-based such as code that is commented out or more specific issues such as unreachable code. Despite this, CodeCheck is limited to only detecting issues specific to a target programming language such as variable use amongst functions or improper pointer use [32]. Architecture-specific bugs like improper communication amongst features cannot be detected because feature communication is not specific to a particular language. The other major feature is the

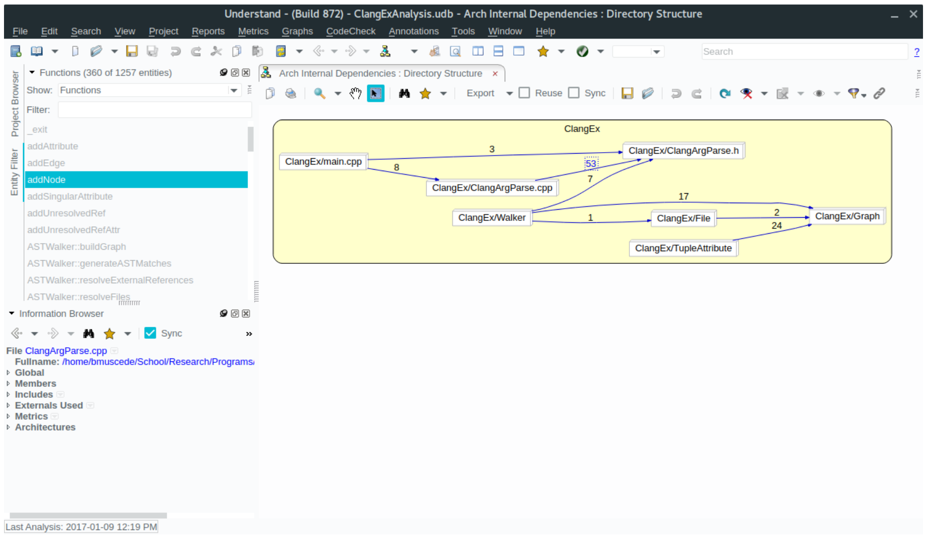
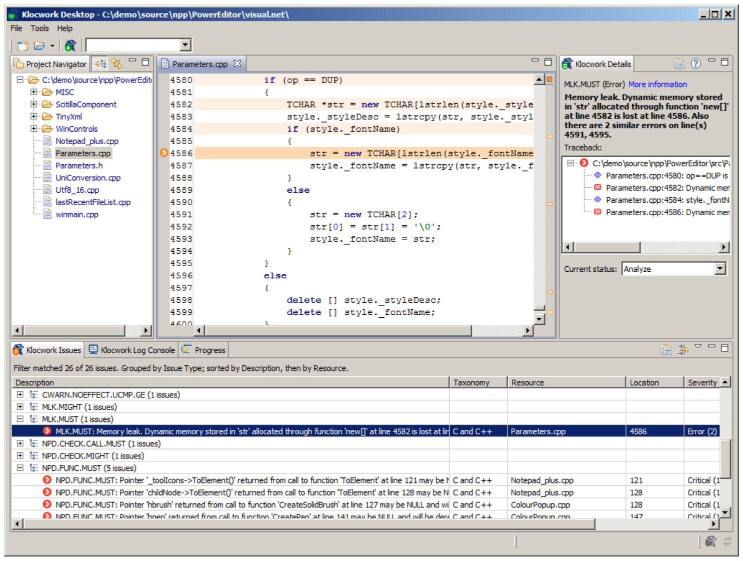

Understand - (Build 872) - ClangExAnalysis.udb - Arch Internal Dependencies : Directory Structure
File Edit Search View Project Reports Metrics Graphs CodeCheck Annotations Tools Window Help
Functions (360 of 1257 entities)
Show: Functions
Filter:
_exit
addAttribute
addEdge
addNode
addSingularAttribute
addUnresolvedRef
addUnresolvedRefAttr
ASTWalker::buildGraph
ASTWalker::generateASTMatches
ASTWalker::resolveExternalReferences
ASTWalker::resolveFiles
Information Browser
Sync
File ClangArgParse.cpp
Fullname: /home/bmuscede/School/Research/Programs
Global
Members
Includes
Externals Used
Metrics
Architectures
Last Analysis: 2017-01-09 12:19 PM
Arch Internal Dependencies : Directory Structure
Export
Reuse
Sync
ClangEx
ClangEx/main.cpp
ClangEx/ClangArgParse.h
ClangEx/ClangArgParse.cpp
ClangEx/Walker
ClangEx/File
ClangEx/Graph
ClangEx/TupleAttribute
3
8
53
7
17
1
2
24

Figure 2.3: The Understand GUI analyzing the ClangEx fact extractor.

ability to generate diagrams that assist users with understanding the interactions between software components. Understand can generate UML diagrams by performing a syntactic scan of the code and can generate dependency digraphs showing which modules interact with each other. Although Understand is valuable, limitations with CodeCheck for detecting feature communications make it difficult to discovering potential feature interactions in automotive software.

Klocwork

Klocwork is a static-analysis tool for C, C++, C#, and Java projects. It is a commercial tool that consists of a variety of different features including static-code analysis, code-metrics reporting, and security-violation detection [33]. Both the static code analysis and security violation detection tools come with a collection of pre-identified fingerprints that Klocwork checks for. These fingerprints are synonymous with hotspots; they are areas of code that have the potential for causing problems. Further, both Klocwork tools allow for users to develop their own fingerprints. These custom fingerprints examine the abstract syntax tree (AST) of the source code and match code segments based on some XPath-like expression. Figure 2.4 shows the Klocwork desktop GUI running on an example project. The right portion of the window shows some error that Klocwork identified.



```
4580            if (op == DUP)
4581            {
4582                TCHAR *str = new TCHAR[lstrlen(style._style
4583                style._styleDesc = lstrcpy(str, style._styl
4584                if (style._fontName)
4585                {
4586                    str = new TCHAR[lstrlen(style._fontName
4587                    style._fontName = lstrcpy(str, style._f
4588                }
4589                else
4590                {
4591                    str = new TCHAR[2];
4592                    str[0] = str[1] = '\0';
4593                    style._fontName = str;
4594                }
4595            }
4596            else
4597            {
4598                delete [] style._styleDesc;
4599                delete [] style._fontName;
```

MLK.MUST (Error) More information

Memory leak. Dynamic memory stored in 'str' allocated through function 'new[]' at line 4582 is lost at line 4586. Also there are 2 similar errors on line(s) 4591, 4595.

Traceback:

```
C:\demo\source\npp\PowerEditor\src\P
    Parameters.cpp:4580: op==DUP is
    Parameters.cpp:4582: Dynamic mer
    Parameters.cpp:4584: style._fontN
    Parameters.cpp:4586: Dynamic mer
```

Current status: Analyze

Filter matched 26 of 26 issues. Grouped by Issue Type; sorted by Description, then by Resource.

Description	Taxonomy	Resource	Location	Severity
CWARN.NOEFFECT.UCMP.GE (1 issues)				
MLK.MIGHT (1 issues)				
MLK.MUST (1 issues)				
MLK.MUST: Memory leak. Dynamic memory stored in 'str' allocated through function 'new[]' at line 4582 is lost at lir	C and C++	Parameters.cpp	4586	Error (2)
NPD.CHECK.CALL.MUST (1 issues)				
NPD.CHECK.MIGHT (1 issues)				
NPD.FUNC.MUST (5 issues)				
NPD.FUNC.MUST: Pointer '_toolIcons->ToElement()' returned from call to function 'ToElement' at line 121 may be N	C and C++	Notepad_plus.cpp	121	Critical (1
NPD.FUNC.MUST: Pointer 'childNode->ToElement()' returned from call to function 'ToElement' at line 128 may be N	C and C++	Notepad_plus.cpp	128	Critical (1
NPD.FUNC.MUST: Pointer 'hbrush' returned from call to function 'CreateSolidBrush' at line 127 may be NULL and wi	C and C++	ColourPopup.cpp	128	Critical (1
NPD.FUNC.MUST: Pointer 'hpen' returned from call to function 'CreatePen' at line 141 may be NULL and will be dere	C and C++	ColourPopup.cpp	147	Critical (1

Figure 2.4: The Klocwork desktop GUI running on an example project [3].

Although Klocwork is powerful, it is not without limitations. First, Klocwork is only able to detect issues with how language elements in a project are used. This limitation means that it would be difficult to express feature-interaction hotspot patterns by using just language constructs. Because many feature-interaction hotspots pertain to architectural patterns such as variable assignment flow or data dependencies, Klocwork would not be well-suited for feature-interaction detection. Further, despite Klocwork having an extensive collection of built-in security checks, many of these checks are unable to discover all violations of that type. This is more of a problem with static analysis as a whole as most static-analysis approaches have the potential of producing false positives and negatives.

2.2 Relational Analysis Toolchain

The relational algebra toolchain is a generic static-analysis toolchain developed at the University of Waterloo. This toolchain is made up of several tools that generate a queryable model of the software artifact. In this thesis, the toolchain is used to identify hotspots in an automotive software system. Figure 2.5 shows each tool in the relational analysis toolchain. Gray boxes represent files, blue boxes represent programs, and the single green box represents additional programs that could be added to the toolchain.

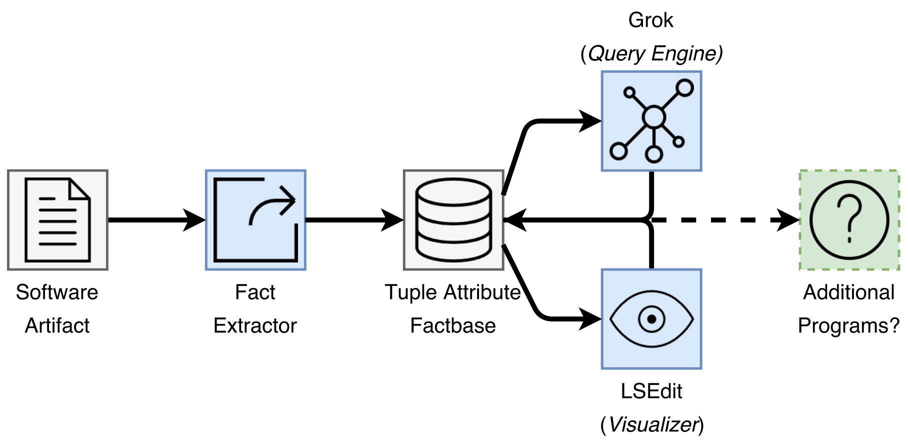


Figure 2.5: The relational algebra toolchain.

The basic idea of this toolchain is to generate a queryable model that represents some number of software artifacts. Such a model is created automatically by feeding software artifacts into a custom, human-defined analyzer called a **fact extractor**. The model generated by a fact extractor is a collection of facts about the software artifact, the collection of facts is called a factbase. Facts can be any desired information about the software artifact: functions, the relationship between a class and its functions, or variables all could be considered facts. Facts are represented in a common, plain-text format known as tuple-attribute (TA). A fact extractor is designed to extract only "important" facts. What is deemed important varies depending on what one wants know about the underlying software. For instance, if a user is interested only in examining the function calls of a program, the queryable model of that program's source code should include just the function call graph. By extracting just the information that is needed from the software artifact, the resultant model's complexity is reduced and queries executed on the model will be faster. Once a model is created, the toolchain has two custom tools called **Grok** and **LSEdit** that allow the user to query and visualize their custom model.

This toolchain is an open framework. New fact extractors can be created to work with other types of software artifacts, new tools other than Grok and LSEdit can be added to read and analyze the factbase. Fact extractors exist for Java [34] and C++ [35] source code files and for ELF object files [36].

The remainder of this section will discuss each portion of the toolchain in detail. Section 2.2.1 will discuss the concept of fact extraction. Section 2.2.2 will discuss the TA format and the concept of factbases. Finally, Sections 2.2.3 and 2.2.4 will discuss two important tools that operate on TA models: *Grok* and *LSEdit* respectively.

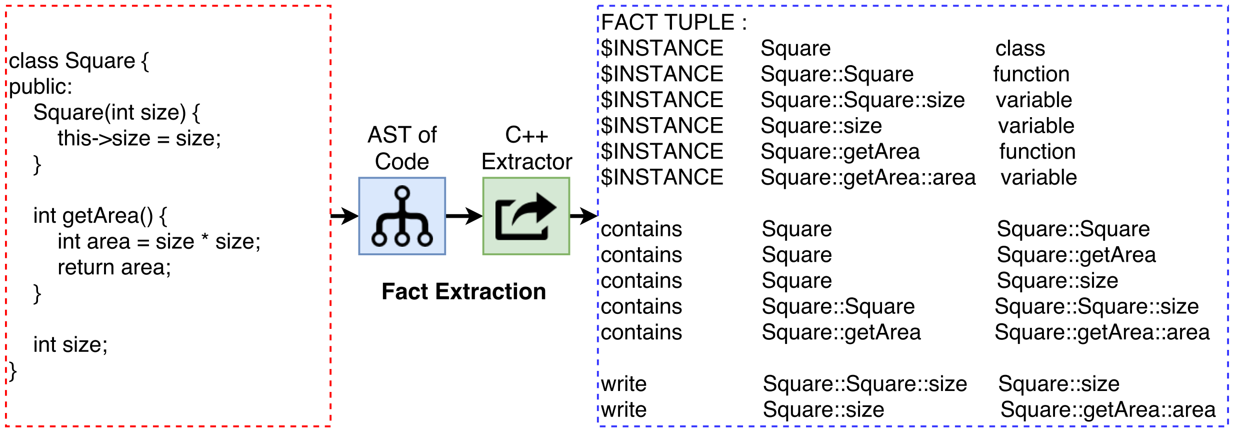
2.2.1 Fact Extractors

Fact extractors are custom, human-defined analyzers that automatically scan *structured* software artifacts to pull out pertinent details to be included in a resultant factbase. Because a fact extractor is tuned to pull out specific information about a particular type of artifact, separate fact extractors have to be created for each programming language one wants to target. Further, different fact extractors may need to be developed for the same language depending on the queries one might want to ask. For instance, there might be one C++ fact extractor that extracts only the function call graph of a program whereas another C++ extractor that might exist to extract all static variables.

Although it may appear daunting to develop a fact extractor for each type of software artifact and for different query types, the hardest part of developing these extractors is determining the *type* of information one wants to extract and how that information should be best exploited. Once this information is collected, fact extractors can be tuned to detect and parse these facts from the underlying artifact. How fact extractors collect information from software artifacts is extremely variable and depends on the type of artifact and the type of information a user wants to collect. Extractors can use debug information from compiled programs [37], the AST of parsed source code, or even information from XML log files [1]. Further, in the case of source code, there is no need to directly parse the source files; object files or executables compiled from the source can be used [38] though the information they collect is not sufficient for feature-interaction detection.

Figure 2.6 illustrates an example fact-extraction process. Here, a fact extractor is used that gleans information from the AST of C++ source code to generate a factbase. This example fact extractor extracts model tracks C++ classes, functions, and variables as well as the relationships that show which entities are "contained" in other entities and the flow of data between variables. The program to be analyzed consists of a simple class called `Square` that represents square objects (shown on left of Figure 2.6). When this program is processed by this example fact extractor, the resultant TA model is generated (shown on the right). This resultant factbase includes several entities describing the functions, variables, and classes present as well as two relations: `contains` and `write`. This creates a factbase describing which functions and variables belong to which class and which variables are written to by which other variables. For instance, the line

[1]XML extraction is used in a version of the Rex extractor not discussed in this thesis.



```
class Square {
public:
    Square(int size) {
        this->size = size;
    }

    int getArea() {
        int area = size * size;
        return area;
    }

    int size;
}
```

```
FACT TUPLE :
$INSTANCE     Square                    class
$INSTANCE     Square::Square            function
$INSTANCE     Square::Square::size      variable
$INSTANCE     Square::size              variable
$INSTANCE     Square::getArea           function
$INSTANCE     Square::getArea::area     variable

contains      Square                    Square::Square
contains      Square                    Square::getArea
contains      Square                    Square::size
contains      Square::Square            Square::Square::size
contains      Square::getArea           Square::getArea::area

write         Square::Square::size      Square::size
write         Square::size              Square::getArea::area
```

Figure 2.6: The conversion of a C++ class to a TA factbase.

`write Square::size Square::getArea::area` says that `Square::size` writes data to the `Square::getArea::area` variable.

2.2.2 Factbases and the Tuple-Attribute Language

	Section Name	Section Header
Scheme	Tuple Schema	SCHEME TUPLE
	Attribute Schema	SCHEME ATTRIBUTE
Fact	Tuple Facts	FACT TUPLE
	Attribute Facts	FACT ATTRIBUTE

Table 2.1: The different sections of a TA file.

Once extracted, facts are stored in a factbase in a format known as tuple-attribute (TA) [39]. These factbases are a collection of facts where each fact is a triplet encoded in a format similar to the Rigi Standard Format (RSF) [39]. As such, a triplet is of the form `<RELATION> <ORIGIN> <DESTINATION>` where `RELATION` is the relation that the tuple belongs to. As such, all relations are binary since each relation type will have a collection of entries of the form `<ORIGIN> <DESTINATION>`. This representation enables factbases to capture facts and relationships that are structured as hierarchical graphs. These facts can then be transformed through a series of relational operators as defined by Tarski's relational algebra [40]. These transformations produce new facts about the system under analysis, and can be added to the factbase.

Factbases encoded in the TA language are written in plain text and are divided into two major sections; tuples and attributes. Tuples describe entities and relationships between entities, and attributes describe the attributes for entities and relationships. Each of these sections have two associated subsections; the scheme and facts. The scheme allows users to define a schema for tuples and attributes and the fact section is where the tuple and attribute instances are stored. This scheme is powerful as it defines **how** entities, relations, and attributes are structured. Table 2.1 shows each section of a TA file along with its purpose.

In a TA file, each of these four sections needs to be preceded by a special keyword that is declared before any element of that section. Figure 2.7 shows an empty TA file with each of the four sections defined. The remainder of this section describes the format of the tuple and attribute sections.

```
SCHEME TUPLE :
//The schema for tuples.

SCHEME ATTRIBUTE :
//The schema for attributes.

FACT TUPLE :
//The entities and relationships.

FACT ATTRIBUTE :
//Attributes for entities and relationships.
```

Figure 2.7: An empty TA file with the sections defined.

Fact Tuples

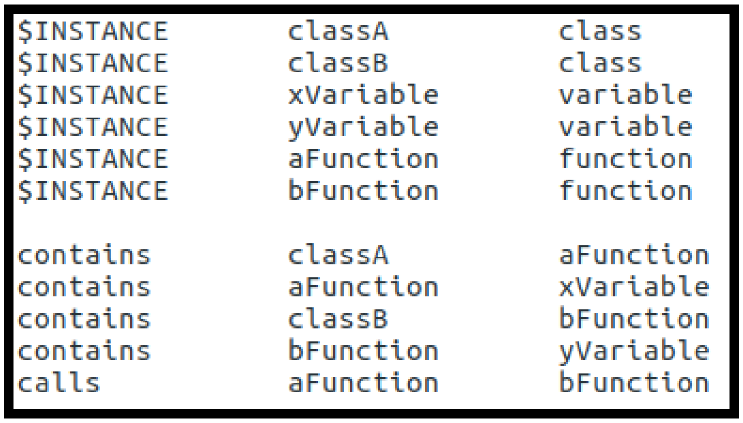
The tuple section of a TA factbase is where entities and the relationships between entities are defined. Tuples follow the Rigi Standard Format (RSF) encoding where they are triplets [41] of the form: `Item1 Item2 Item3`. Both entities and relationships are written in this RSF format.

Entities are of the form: `$INSTANCE <Item_Name> <Set_Name>`. The `$INSTANCE` flag indicates that this triplet is an entity. The `<Item_Name>` is the unique ID for that entity.

The ID must not contain special symbols like spaces or colons. Lastly, `<Set_Name>` is the name of the set that this entity belongs to. By grouping entities into sets, relational queries can target specific groups of entities all at once. For instance, one could have a set called `Variables` that groups all variables in a C++ program together.

Relationships are encoded in a similar fashion as: `<Relationship_Name> <Item_1> <Item_2>`. The `<Relationship_Name>` is the type of the relationship that this entry is part of. This is similar to the `<Set_Name>` field for entities. The `<Item_1>` and `<Item_2>` are the IDs for the source and destination entities.

Figure 2.8 shows an example of tuples encoded in a TA factbase. Here, there are six different entities and five different relationships. There are also three types of entities: `class`, `variable`, and `function`. Also, each of these entities participates in some relationship. For instance, `classA` contains `aFunction`.



```
$INSTANCE       classA          class
$INSTANCE       classB          class
$INSTANCE       xVariable       variable
$INSTANCE       yVariable       variable
$INSTANCE       aFunction       function
$INSTANCE       bFunction       function

contains        classA          aFunction
contains        aFunction       xVariable
contains        classB          bFunction
contains        bFunction       yVariable
calls           aFunction       bFunction
```

Figure 2.8: An example of TA tuples for a generic C++ program.

Fact Attributes

The attribute section describes attributes for previously defined entities and relations in the TA model. Attributes are untyped key-value pairs that can either be unique for a single entry or applied to the entire entity or relationship set [42]. There are two types of attributes: single-value and multiple-value attributes.

Attributes are defined in the `FACT ATTRIBUTE` section for each entity or relationship. Before attributes are written, the entity or relationship must be previously-defined in the

TA file. To define attributes on an entity, the entity ID is written first and to define attributes on a relation, the entire relationship string is written surrounded in brackets. Then, a list of attribute key and value pairs are written surrounded by curly braces. Single-valued attributes take the form *key* = *value* and multiple-valued attributes take the form *key* = (*value1 value2* ...).

```
FACT ATTRIBUTE :
aFunction { label = classA::aFunction isStatic = 1 }
bFunction { label = classB::bFunction isStatic = 0 }
xVariable { linesUsed = (5 10 15) }
(calls aFunction bFunction) { numberOfCalls = 5 }
```

Figure 2.9: Examples of TA attributes.

Figure 2.9 shows several example lines of the attribute section of the TA file. In it, there are several entities and one relationship. For each entity, `aFunction` and `bFunction`, there is a label attribute and an isStatic attribute. For the `xVariable` entity, there is a multiple-valued attribute for the lines in which it is used. The calls relation between `aFunction` and `bFunction` has an attribute called numberOfCalls.

2.2.3 Relational Query Engine - Grok

Grok is a relational algebra calculator that allows users to execute relational algebra queries on TA factbases [27]. With the Grok tool, a user loads in a TA factbase, executes relational operations on the sets and relations in the factbase, and then adds the resulting new facts to the factbase [42]. Table 2.2 show a subset of the relational operations that can be performed[2]. Grok can be run interpretively through the command line or can be invoked using scripts that execute certain relational commands and output the results.

Grok was originally created as a program comprehension tool [42] to allow users to write queries that would allow users to infer the architectural structure of a software system, improving their comprehension of that system. Since then, the use of Grok has expanded to other problems such as clone detection in source code [43].

In addition to Grok, a Java-based version called jGrok was developed in 2006 [38]. Although jGrok has a similar syntax to Grok and supports the same operations, there

[2]Other Grok operations and syntax can be found at `http://www.swag.uwaterloo.ca/grok/grokdoc/index.html`

Relational Operation	**Grok Operator**	**Description**
Union Operator	<ITEM 1> + <ITEM 2>	Can be performed on two sets or two relations. Merges entries contained both in <ITEM 1> and in <ITEM 2>.
Intersection Operator	<ITEM 1> ˆ<ITEM 2>	Can be performed on two sets or two relations. Only keeps entries from <ITEM 1> and <ITEM 2> if it exists in both.
Difference Operator	<ITEM 1> − <ITEM 2>	Can be performed on two sets or two relations. Removes entries in <ITEM 1> that are also contained inside <ITEM 2>.
Composition	<ITEM 1> o <ITEM 2>	Joins two entries together. Either joins two sets or a set and a relation. The join occurs on the rightmost entity of the first item and leftmost entity of the second item.
Transitive Closure	<RELATION>+	Computes the transitive closure of a relation. This determines whether items in the relation are reachable from other items.
Cross Product Operator	<SET 1> X <SET 2>	Creates a relation consisting of all possible combinations of entries from <SET 1> and <SET 2>. This computes the cross product of both sets.

Table 2.2: A subset of supported Grok operations for TA factbases.

are some subtle differences. The benefit of jGrok is that it supports several additional operations such as better math and file operations. However, one drawback is that jGrok is slower compared to Grok due to fewer performance optimizations and because it runs in the JVM.

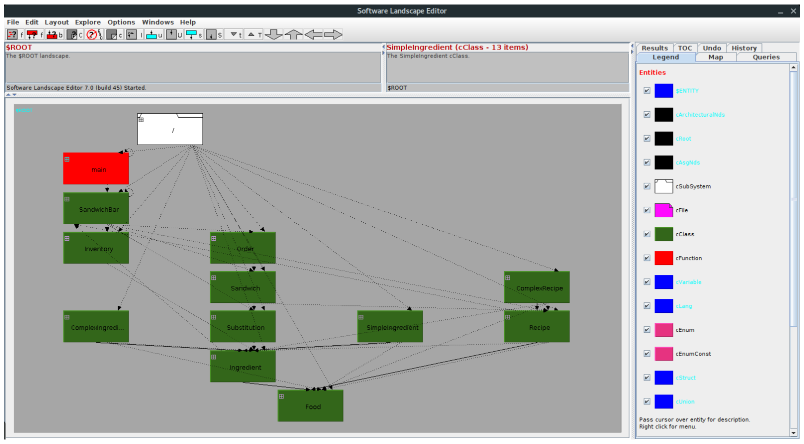


Figure 2.10: LSEdit visualizing an example project with several classes.

2.2.4 Model Visualizer - LSEdit

The Landscape Editor (LSEdit) is a tool that operates on a TA factbase to visualize, manipulate, and cluster hierarchical graphs [44]. The initial purpose of LSEdit was to provide a means to easily visualize architectural models of large software systems by using the semantics from TA models to perform node clustering through hierarchies. Figure 2.10 shows an example of LSEdit visualizing an example model. Here, entities are shown as coloured boxes and relationships are shown as lines between the boxes. In this particular visualization, relationships between C++ classes (green).

To support complex visualizations, LSEdit makes use of the scheme and attribute sections of TA factbases. LSEdit uses "special" attributes defined on nodes that tell LSEdit how to format them. Node colours, display names, and appearance can all be modified by setting specific attributes for each node type. Further, by default, LSEdit obtains hierarchy information by looking for a `contain` relation in the TA file. Children nodes "contain"ed inside parent nodes will be displayed as a hierarchy in the resulting visualization. This hierarchy relation can be set to be any relation in the TA file.

2.3 Robot Operating System (ROS)

The Robot Operating System (ROS) is a popular, programming framework that is designed to assist programmers with developing robotic applications. This framework is used in nu-

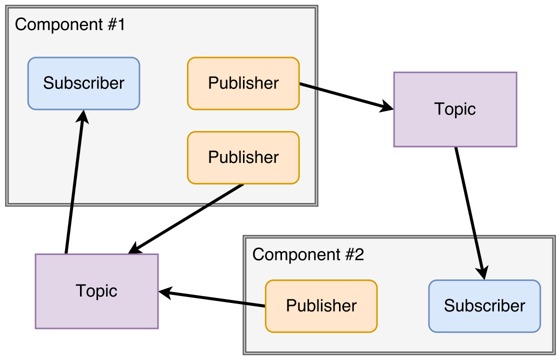


Figure 2.11: An example of the publisher-subscriber paradigm in ROS.

merous autonomous robotic applications including the Autonomoose self-driving platform. It is important to understand how ROS works because the Autonomoose project is used in a case study in Chapter 5 to evaluate the effectiveness of the relational algebra toolchain in hotspot detection.

Essentially, ROS is a collection of C++ libraries that provide several services for robotic developers. Although ROS is named as an operating system, it is not an operating system in the traditional sense. Rather, ROS provides services that assist programmers in developing robotic systems that run on heterogeneous computing clusters. These services include hardware abstraction, inter-process communication, and package management [45]. ROS is fairly common in the robotics industry and there are numerous hardware systems that support ROS. For instance, iRobot, the maker of the popular Roomba autonomous vacuum, sells a modified, hackable version the vacuum that is designed to run ROS [46].

The remainder of this section describes the communication framework of ROS. Although ROS has other services, Autonomoose makes heavy use of the communication framework. Understanding the ROS communication framework is important to understanding the hotspots described in Chapter 4 and the case study described in Chapter 5.

2.3.1 Communication Framework

Robotic systems tend to consist of multiple Electronic Control Units (ECU) distributed across the robotic chassis. These ECUs need to be able to send messages between each other to carry out actions. To carry out message-passing, ROS uses an event-driven, publisher-subscriber architecture where components can publish and subscribe to a named topic. When a component wants to send data about particular information, it publishes that data to some relevant topic. Then, any node that is subscribed to that topic will receive the data. There can be zero-to-many publishers and zero-to-many subscribers for each topic. Additionally, publishers and subscribers are independent; they are unaware of other publishers or subscribers connected to their topic. Figure 2.11 shows an example of two ROS components communicating with each other using this publisher-subscriber system. In this figure, note that Component 1 and Component 2 both have publishers that publish to the same topic.

Although Figure 2.11 shows topics as an intermediate buffer that exists outside of each component, topics are simply named buses on which messages flow [45]. ROS facilitates sending a message from a component's outgoing buffer to all subscriber's incoming buffers. When topics are created, users specify the type of data that pass through that topic. These datatypes can be primitive types such as `int` or `double` or more complex types like RADAR data or LiDAR pointclouds. Because message data are not buffered in queues outside of components, each publisher or subscriber that connects to a topic bus needs to specify an outgoing message queue size (for publishers) or an incoming message queue size (for subscribers). If message queues fill up, the ROS framework specifies that old messages will be dropped. Message queues can also be infinite; however, this could cause an error condition where publishers flood a subscriber with messages causing it to crash.

2.4 Chapter Summary

In this chapter we presented several related approaches including JITANA and work by Purandare et al. that detects communications amongst components in the Android Operating System and Robot Operating System respectively. In addition, we presented the relational analysis toolchain. This toolchain is an abstract process that generates a queryable factbase from software artifacts. These factbases can be manipulated through the use of relational algebra operations to derive new facts. Lastly, we provided an overview of the Robot Operating System framework that is used in the software system tested in the case study in Chapter 5.

Chapter 3

Fact Extractors

As described in Section 2.2, before the relational algebra toolchain can be used to detect hotspots in a software project, a factbase of information about that software needs to be created by running the source code through a fact extractor. The fact extractors need to generate high-quality, accurate models that comprise sufficient information to allow for the detection of hotspots. Fact extractors operate by automatically extracting *important* information about a software artifact. What is deemed important depends on the type of hotspots one wants to extract. Factbases outputted by fact extractors contain a predefined schema that describes the type of entities, relationships, and attributes contained in the factbase.

For this thesis, we developed two different fact extractors; **ClangEx** and **Rex**. Both extractors were developed to extract enough information from C and C++ source code to perform feature-interaction hotspot detection in software systems. Although these two extractors are both capable of detecting feature-interaction hotspots and target the same programming languages, the type of information they extract and the models they generate are different. Each extractor is designed to be effective for different types of situations.

Numerous other extractors have been developed through previous research [34][35][37]. Further, because the relational analysis toolchain was initially developed to help programmers understand a software system, many extractors developed in previous work aim to support this task [47]. Three notable extractors developed prior to this thesis are **CPPX**, **ASX**, and **bfx64**; all extract C and C++ code and were used as an initial starting point for work on the ClangEx and Rex extractors.

CPPX Created by Ric Holt, Tom Dean, and Andrew Malton, CPPX is the first C++ fact extractor developed [35]. It utilizes a modified version of the GNU compiler (`gcc`) to gain access to the abstract syntax tree (AST) and, from it, obtain information about the underlying source code. CPPX was crucial in the development of ClangEx and Rex as it tackled numerous issues that come with analyzing C++ source code. This includes issues such as resolving header declarations with source definitions. Although we used some of CPPX's solutions, ClangEx and Rex use the Clang compiler rather than `gcc` to gain access to the AST. Using Clang allowed for easier AST access and well-defined API calls to gather source code information.

ASX Created by Ian Davis, ASX is a fact extractor that targets C, C++, and assembler code[1] [37]. By reverse engineering debugging information, ASX is able to gather information about functions, variables, classes, and item usage. Despite ASX being useful for assisting developers with program comprehension, it has also been used in other research to detect and eliminate dead functions in a software project [48]. ASX was important in developing ClangEx as we used it as a "sanity" check to compare results between it and ClangEx. Further, the ASX schema for TA files was used as a starting point to decide upon the entities to include in the ClangEx model.

bfx64 We developed this extractor while conducting initial research into the relational algebra toolchain for use in hotspot detection. This extractor is based on the original BFX extractor developed by Jingwei Wu [38] and targets ELF object files. Although bfx64 is capable of targeting C and C++ code, the type of information that can be gathered from object files is not detailed enough for feature-interaction hotspot detection because only information about functions and variable is available. As bfx64 is unable to generate these types of models, work was started on ClangEx and Rex. However, algorithms from bfx64 were reused in ClangEx and Rex, such as the algorithm to covert the in-memory hierarchical graph to TA.

Table 3.1 compares the bfx64, ClangEx, and Rex extractors. The fact extractor names in bold highlight the extractors discussed in this thesis. The *Language(s)* column indicates the source programming language each extractor targets. The *Compilation Level* column indicates how each extractor obtains information about the source code. For instance, bfx64 obtains its information to generate its models by deconstructing object files. Finally,

[1]To operate, ASX utilizes version 5 of the DWARF debugging format standard. Programs compiled with future versions of the DWARF standard may not be compatible.

Comparison of Fact Extractors			
Name	*Language(s)*	*Compilation Level*	*Information Extracted*
bfx64	C, C++	ELF Object Files	Captures function calls and relationships between functions and variables.
ClangEx	C, C++	Abstract Syntax Tree	Captures basic C and C++ language information such as function calls, variable reads and writes, and class information.
Rex	C, C++	Abstract Syntax Tree	Captures ROS publisher and subscriber information, function calls, variable reads and writes, and control-flow information.

Table 3.1: A comparison of several publically available fact extractors.

the *Information Extracted* column indicates the type of information each extractor gathers from the input source code.

The remainder of this chapter introduces the ClangEx extractor in Section 3.1 and the Rex extractor in Section 3.2. For each extractor we present the motivation behind its development, its tuple-attribute schema, and its internals. Further, each section discusses the benefits and drawbacks of using the extractor. Although not a part of this chapter, Appendix A provides installation and usage instructions for the ClangEx and Rex extractors and Appendix B provides installation and usage instructions for the bfx64 extractor.

3.1 ClangEx

ClangEx (**Clang Ex**tractor) is a C and C++ extractor that uses the Clang open-source compiler to generate a model of a software project by parsing the underlying source code. Specifically, ClangEx analyzes abstract syntax tree (AST) fragments created during the compilation of C and C++ source code, via an extensive API and allows ClangEx to operate on the C/C++ AST. By having access to the AST, ClangEx is able to obtain a large amount of information about the source code it is analyzing. This extractor was developed with no particular goal in mind; it serves as a simple generic fact extractor that can extract information about C and C++ language features like classes, functions, and variables. By extracting only AST-based elements from source code, ClangEx serves as a model extractor that can easily be extended to achieve specific goals. For instance, the Rex extractor (discussed in Section 3.2) is a modified version of ClangEx that can detect ROS messages between components. Though ClangEx is not used in the case study presented in Chapter 5, the purpose of describing it in this thesis is to highlight its design and show how it was extended to build Rex. Other extractors that collect message-passing information could also be built by extending ClangEx.

Due to the idiosyncrasies of processing C and C++, ClangEx has two main analysis modes: (1) regular mode; and (2) full-analysis mode. The purpose of these two modes is to utilize two different source-code processing methodologies. Although both modes use the schema shown in Figure 3.1, models generated by the two approaches can be drastically different due to how the modes deal with header files. In regular mode, ClangEx examines all features inside a C or C++ source file and ignores header files. This ensures that any unnecessary declarations contained in header files but not directly related to the source file are not included in the model. In contrast, in full-analysis mode, ClangEx looks at all language features inside the source file and drills down into the contents of all included header files (excluding system header files). This produces a bloated model because code fragments from other source file's headers will be included in the model even if those source files are not directly selected for analysis. The preferred processing mode depends on the intended queries. For instance, if a user is analyzing the whole project, full-analysis mode is recommended because the extractor will encounter all definitions for all header file declarations. Note that if a function is declared and defined inside a header file, then ClangEx in regular mode will not include that function in the model whereas full-analysis mode will.

3.1.1 ClangEx Metamodel

Figure 3.1 shows the metamodel that describes all models created using the ClangEx extractor. This section presents terminology used and then describes each section of the metamodel.

Terminology

Before the metamodel is described, some terminology used in the metamodel is presented. First, the concept of a *language feature* is introduced. Language features are code fragments that are specific to a particular programming language. As described in Section 2, when building a fact extractor, users need to decide on the language features they want to extract. These language features are highly dependent on the language. For instance, they could be `try` statements, `variable` declarations, or `statements`. TA models produced using a fact extractor will record *instances of language features* detected in source files. There could be zero or many instances of a particular language feature in a source file. These instances are commonly known as entities in the TA file.

Whether certain language features are recorded as nodes, relationships, or attributes depends on the type of language feature and how the metamodel is designed. For instance, `variable`s are recorded as a node whereas statements like `varA = varB` are recorded as relationships.

Metamodel Elements

At its core, the ClangEx metamodel describes two different classes of nodes. First, it records nodes that come from language features specific to the C and C++ programming languages. For the language-feature class, there are specific nodes that are recorded including `variables`, `functions`, `enums`, `structs`, and `unions`. Each language-feature instance detected in source code is recorded as a node in the TA factbase, has various relationships that it partakes in, and also has numerous attributes that are specific for that type of language feature. For instance, a `function` node has attributes that describe whether it is static, volatile or variadic and a `variable` node has attributes that describe its scope. In Figure 3.1, attributes recorded for each node type are written underneath the name of that node. One attribute common to all node types that describe a language feature is the *filename* attribute. This attribute describes the files that node is declared or defined in. If a function is declared inside one header file and defined inside a source file, the *filename* attribute will store both filenames.

ClangEx also stores information about the relationships in which each of these language node types partake. Overall, there are several main relationship types: `contain`, `references`, `inherits`, and `calls`. A `contain` relation is used whenever ClangEx wants to indicate that an entity is contained inside another entity. For instance, a `class` can contain `functions` or `variables`. This relationship is many-to-one meaning that a node being contained inside some container node cannot also be inside any other containers. A `reference` relation indicates when some entity refers to another entity. For instance, a `function` may refer to a `variable` or `field`; this relationship is a “catch-all” for the types of interactions that occur between entities. The `inherits` relation is used whenever a `class` inherits from another `class`. Lastly, the `calls` relation is explicitly reserved for

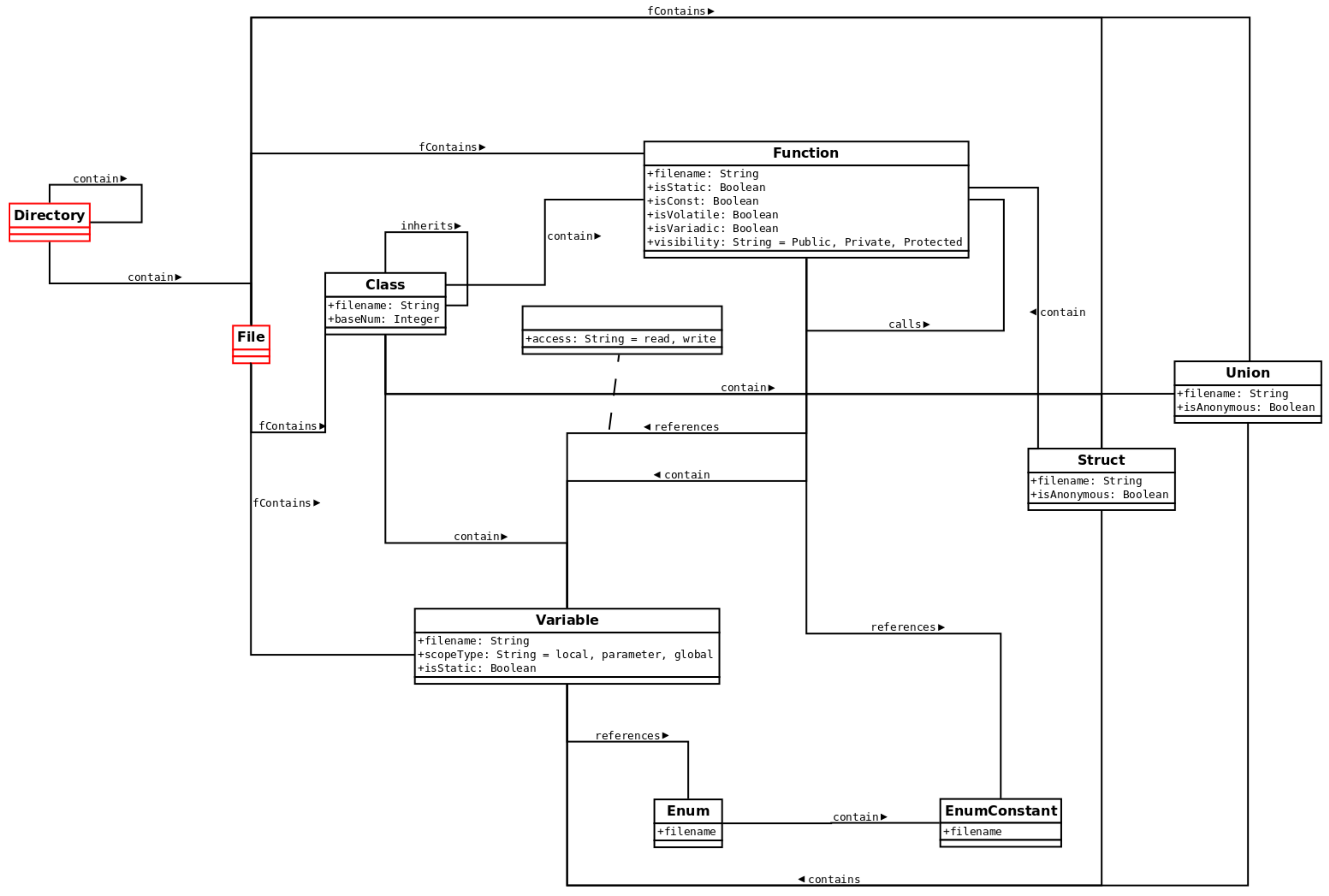


Figure 3.1: The ClangEx metamodel. Black classes are C/C++ language features.

representing function calls. This relationship represents the call graph for the project that the model represents.

In addition to nodes that describe instances of C or C++ language features, models produced using ClangEx also contain nodes that describe files and directories. File nodes are facts that typically represent source or header files. ClangEx extracts facts about language features that appear in files , but does not use the `contain` relation to link files and C/C++ language features together because a node can only be "contained" inside one container at a time. Instead, for each C and C++ language feature node (`struct`, `class`, etc.), there is one or more special `fContain` (file contain) relations that point from some file node to that C/C++ node.

3.1.2 Advantages of the ClangEx Extractor

Despite the fact that other C and C++ extractors exist [35], ClangEx has several advantages. First, using the Clang API, ClangEx is able to easily obtain detailed information about source code easily. Unlike previous extractors which use heavily modified compilers to conduct extraction, using the Clang API allows ClangEx to read AST elements as Clang parses the source code. As such, adding new C/C++ elements to extract is easier with Clang because the Clang API already provides methods to access this information.

As ClangEx obtains information from the AST of a source file, ClangEx does not require a project to be linkable or even fully compilable. Thus, if a project has compiler errors or if included libraries are missing, Clang will still generate a partial AST that can be read by ClangEx. This means that a user can analyze a subset of the project. The extractor performs its own linking of equivalent references from separate compilation units. If a reference cannot be resolved by the end of the extraction process, it is determined that the reference is not part of a file that was passed to ClangEx, and, ClangEx simply deletes any relationship that uses that reference from the factbase.

3.1.3 Disadvantages of the ClangEx Extractor

A major disadvantage is that ClangEx is bound to the Clang compiler. Although this may not appear to be an issue, Clang is not able to compile all C/C++ programs out of the box. In particular, Clang is unable to process non-standard C or C++ code. For instance, the `gcc` compiler has a collection of "gcc-isms" that are not part of standard C but are supported by `gcc` [49]. Any projects that take advantage of the "gcc-isms" cannot be analyzed by ClangEx. The Linux kernel is one such project that suffers from this problem; the Linux 3 kernel uses `asm goto` statements that Clang cannot process.

Second, ClangEx can not be easily incorporated into existing build workflows. ClangEx utilizes a Clang compilation database to specify the compiler flags for each file. This database is a JSON database that lists each source file for a project as well as all compilation flags required to build that file. Although build workflows that use CMake can easily generate compilation databases through the `COMPILE_COMMANDS` CMake flag, workflows that use `make` need to use external tools (such as the Bear tool[2]) to generate these databases. As such, it may take some effort to analyze a large-scale project with ClangEx because the whole project will need to be compiled in advance to generate a compilation database prior to using ClangEx.

3.1.4 ClangEx Internals

As previously discussed, ClangEx makes heavy use of the Clang API to gather AST information from a collection of source code files. Thus, ClangEx does not parse or traverse the AST itself; this is handled by Clang. Only when Clang encounters a C or C++ language feature of interest does Clang pass control of the AST over to ClangEx.

ClangEx is divided into three major sections that are responsible for generating a TA model from source code 1. the Driver module that carries out command-line operations and activates Clang, 2. AST Walker modules that carry out logic to process desired sections of the AST, and 3. the Graph module that maintains an in-memory graph of the source code being analyzed. Figure 3.2 shows how each module operates while a C or C++ source file is processed. The user is able to select either regular mode or full-analysis mode. Regular mode ignores any included header files whereas full-analysis mode processes all non-system header files. The Driver sets up and starts Clang which parses the source code and generates the AST. Each time Clang encounters a portion of the AST that matches some characteristic deemed "desirable" by ClangEx, Clang pauses its source code parsing and hands control over to ClangEx. Then, ClangEx parses the C/C++ declaration or statement, creates a node or edge that describes that item, and adds it to its in-memory hierarchical graph factbase. References to undeclared entries are saved in a pool of undefined references. Once Clang finishes parsing all specified source files, ClangEx attempts to resolve any undefined or unknown references contained in the in-memory hierarchical factbase. If a reference cannot be resolved, that reference is not included in the final factbase. For commands and how to use ClangEx, see Appendix A.

[2] Bear (**B**uild **EAR**) can be found at: https://github.com/rizsotto/Bear

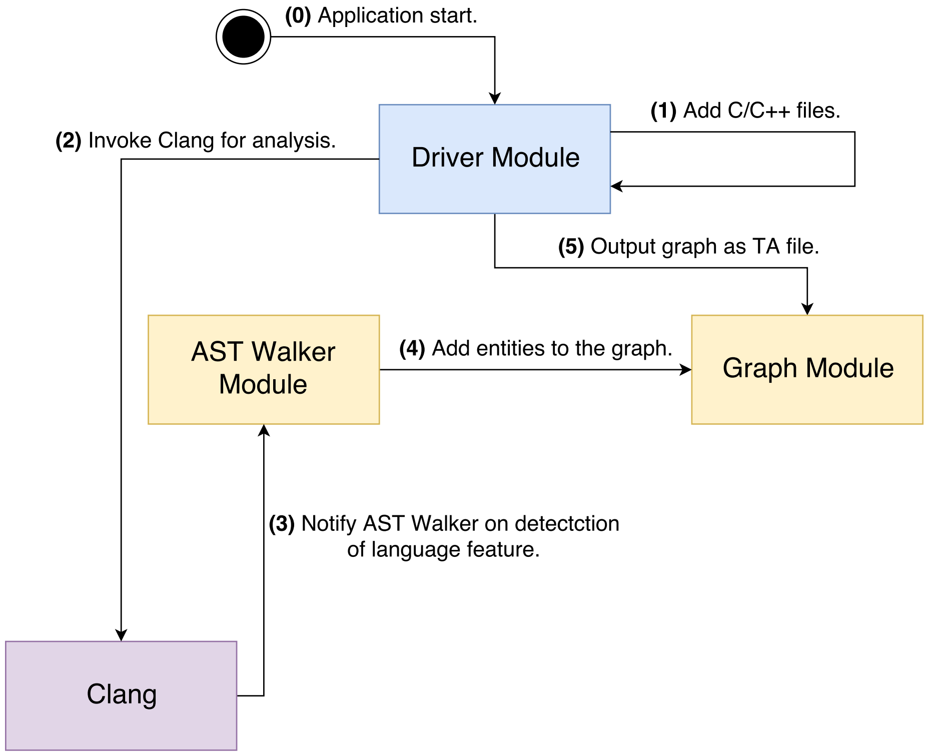

(0) Application start.
Driver Module
(1) Add C/C++ files.
(2) Invoke Clang for analysis.
(5) Output graph as TA file.
AST Walker Module
(4) Add entities to the graph.
Graph Module
(3) Notify AST Walker on detectction of language feature.
Clang

Figure 3.2: The modules and the order of their use in ClangEx.

AST Walker Modules

An AST Walker is a component that "walks" through the AST as it is being processed and runs specific code when specific language elements are encountered. AST Walkers vary in behaviour but all AST Walkers have two specific elements: 1. a section that defines AST matchers that specify the language features that ClangEx is interested in and 2. a function called `run` that is invoked whenever a specific language element is encountered. AST Walkers can be toggled on and off at a user's request.

In the case of ClangEx, there are two AST Walkers a user can select, each corresponding to a different type of analysis mode; regular and full-analysis. Regular mode processes only source files whereas full-analysis mode processes source and included header files. Each of these two Walkers has their own set of AST matchers and their own specialized code which runs when a match is found. Technically, both these walkers process the same elements however the full-analysis walker traverses down AST branches for included header

AST Matchers		
	Full-Analysis Mode	*Regular Mode*
Classes	Matches any class declaration statement.	Does not directly look for classes. Finds the class that each function or variable belongs to.
Functions	Matches both function declarations and definitions.	Only functions defined inside the source file will be included. If a function is declared and defined inside a header file it won't be found.
Variables	Finds all variables and fields as well as their usage.	Finds all variables. Fields that are used will be detected.
Enums	All enums and enum constants will be detected.	If an enum or enum constant is used inside the source file, all associated enum constants will be included.
Structs	All structs and fields inside the struct will be detected. Anonymous structs apply.	If a struct is used inside the source file, all entries inside the struct will be included.
Unions	All unions and fields inside the union will be detected.	If a union is used inside the source file, all entries inside the union will be included.

Table 3.2: Language features supported by ClangEx.

files. In both AST Walkers, the specialized code converts an AST element into a node or relationship that is then added to the digraph maintained by the Graph module.

Table 3.2 highlights the differences between the regular and full-analysis AST Walker for each language feature that ClangEx supports. Essentially, the major difference between the regular and full-analysis AST Walker is that the full-analysis Walker processes code fragments contained inside all header files included in the source file. A second big difference is that full-analysis mode finds all references and regular analysis finds references used only in the source file.

Figure 3.3 shows a portion of the AST matcher code from the full-analysis AST Walker; this specifies which matchers for which AST fragments should be turned on. These matchers detect `functions` and `classes`, and they trigger only when portions of the AST match **exactly** the statements described by the matchers. As an example, take line 8 in Figure 3.3. This matcher detects a C or C++ call expression: it binds a variable to the call-expression AST object and another variable to the AST function object of the function

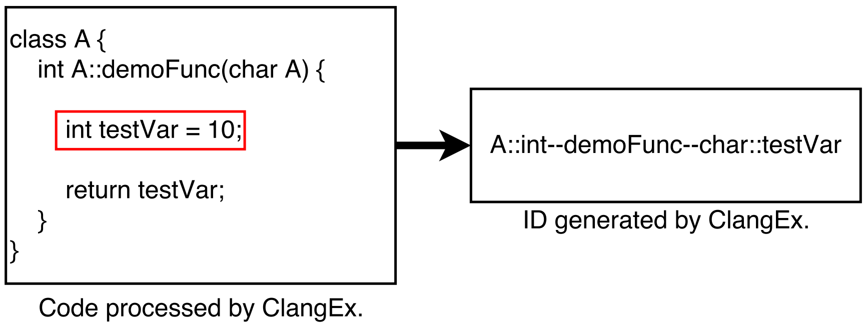
```
void FullWalker::generateASTMatches(MatchFinder *finder) {
    //Function methods.
    if (!exclusions.cFunction) {
        //Finds function declarations for current C/C++ file.
        finder->addMatcher(functionDecl(isDefinition())
                .bind(types[FUNC_DEC]), this);

        //Finds function calls from one function to another.
        finder->addMatcher(callExpr(hasAncestor(functionDecl()
                .bind(types[FUNC_CALLER])))
                .bind(types[FUNC_CALLEE]), this);
    }

    //Class methods.
    if (!exclusions.cClass) {
        //Finds class declarations.
        finder->addMatcher(cxxRecordDecl(isClass())
                .bind(types[CLASS_DEC]), this);
    }
}
```

Figure 3.3: Example matchers for the full-analysis AST Walker module.

where the call originated.


```
class A {
   int A::demoFunc(char A) {

      int testVar = 10;

      return testVar;
   }
}
```

Figure 3.4: An example of ClangEx ID generation for the `testVar` variable.

When an AST matcher is triggered, ClangEx adds the relevant nodes or edges to the hierarchical graph maintained by the Graph module. Adding a graph item involves several tasks. First, Clang generates a unique and linkable ID for the item. This ID must be the same regardless of where this item was encountered, so that references to this item in different compilation units can be linked to the same ID. Developing a unique node ID is not difficult in C, but it can be extremely complicated in C++ due to function overloading. To generate an ID, the Walker gets the locally declared name (not the fully qualified name). If the item being named is a function, ClangEx augments the local name with (i) the function's return type as a prefix and (ii) the types of all the parameters in order of declaration as a suffix. Then, ClangEx prepends the ID of the item's parent node in the AST. This process continues until the top of the AST has been reached. Although C has no concept of fully qualified names or function overloading, ClangEx generates IDs in this manner for both C and C++ artifacts. Figure 3.4 shows the ID that would be generated for the local variable `testVar` declared inside a function. The ID of a generated edge consists of the source node ID combined with the destination node ID.

The Walker must also generate attributes for each item as described by the ClangEx metamodel. To do this, each declaration or statement object has a set of functions, as per the Clang API, that allows ClangEx to gather information about it. This includes its visibility, line number, and the filenames to which it belongs. For each entity, this filename attribute is updated each time that entity is encountered inside a new header or source file not already recorded in the filename list. Once all of this information is collected, ClangEx adds the node or edge to the hierarchical graph maintained by the Graph module.

Once Clang terminates, the Driver module notifies the Walker module that it needs to resolve any undefined references. In the context of ClangEx, undefined references are stored in an unordered list called the edge list. Each entry in the edge list consists of a source node ID, destination node ID, and an edge type. These references describe edges that could not be added because one of the two nodes did not exist when the edge was originally supposed to be added. During this phase, ClangEx iterates through each entry in the edge list and checks the Graph module to see if both node IDs now exist. If they do, ClangEx creates a new edge that corresponds to the information in the edge list and adds it to the digraph. If not, that undefined edge list entry is removed from the list and the graph remains unchanged.

Figure 3.5 gives an example of the edge list being used to resolve undefined references. Starting with an initial graph that contains several function and variable nodes and a single references relation, ClangEx would process the edge list shown in Figure 3.5 and add any relations where the source and destination nodes exist. Here, two edge list entries could be resolved and one could not because the `functionB` function does not exist in the digraph.

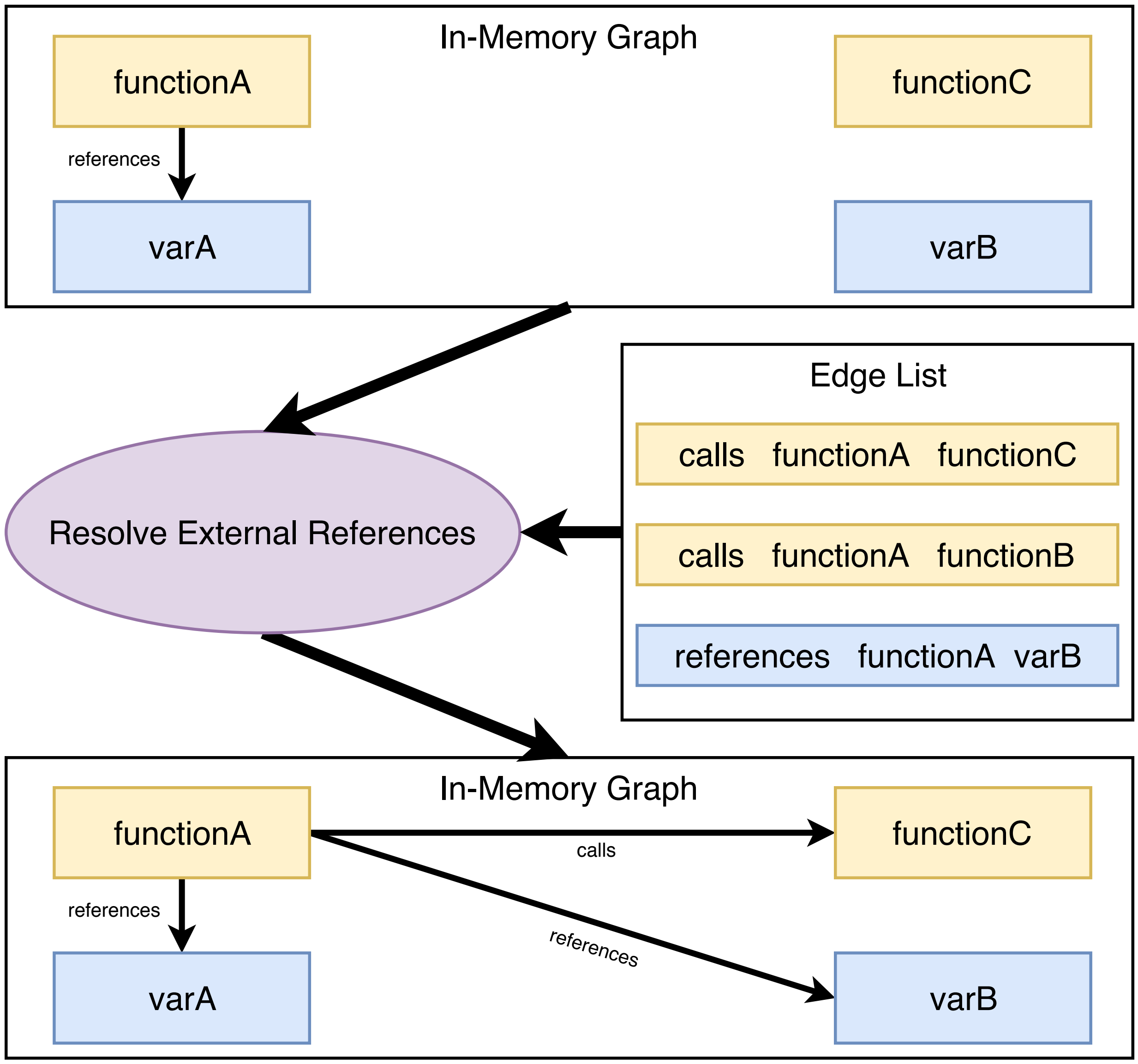


Figure 3.5: An example of how unknown references are resolved after compilation.

Graph Module

The Graph module maintains a digraph of all encountered nodes and edges while ClangEx is running. It stores pointers to nodes and edges in hash maps: nodes are stored with unique IDs as the hash; and edges are stored in two different hash maps, one with source IDs as the hash and the other with the destination IDs being the hash. Although this means that the graph has a space complexity of $\Theta(V + E)$ (where V are the number of vertices and E are the number of edges), looking up a node by ID has a time complexity of $\Omega(1)$ (assuming no collisions) while looking up an edge by source or destination ID has a worst-case time complexity of $\Omega(E)$. The graph module contains numerous getter and setter functions that allow AST walkers to operate on the graph.

In addition to maintaining the graph, the Graph Module is responsible for converting the hierarchical graph into a TA model. Algorithm 1 shows the algorithm responsible for carrying out this conversion. This algorithm starts with a set of nodes called N and a set of edges called E. First, the hierarchical graph's schema is printed. Then, the algorithm prints the nodes and then the edges. For each node in the node set N, a line is generated in the TA file of the following format: `$INSTANCE <NODE_ID> <NODE_TYPE>`. For each edge in the edge set E, a line is generated in the TA file of the following format: `<EDGE_TYPE> <SOURCE_NODE_ID> <DEST_NODE_ID>`. Forward declaration is not allowed in TA so all nodes are declared in the TA file before all edges. Lastly, attributes are then generated for each node and edge that contains at least one attribute. For each node in N and each edge in E with attributes, the following is outputted to the TA file: for nodes the line `<NODE_ID> { <ATTRIBUTE_KEY> = <ATTRIBUTE_VALUE> ...}` is printed and for edges the line `(<EDGE_TYPE> <SOURCE_NODE_ID> <DEST_NODE_ID>) { <ATTRIBUTE_KEY> = <ATTRIBUTE_VALUE> ...}` is printed.

3.2 Rex

Rex (**R**OS **Ex**tractor) is a C and C++ extractor that is designed to capture ROS messages sent between components in a robotic system. As described previously in Section 2.3, ROS is a C++ framework that facilitates the easy communication of components by using a publisher/subscriber architecture. Rex is capable of extracting information about `publishers`, `subscribers`, `topics`, and the messages sent between functions in components via `publish` and `subscribe` calls. To carry out its analysis, Rex is a modified version of ClangEx; it uses the Clang API to gain access to the AST of a source file as it is being parsed. Many of the internals of Rex are similar to those of ClangEx which highlights the

Algorithm 1 Tuple-Attribute Model Generation from Digraph

```
 1: procedure writeTAModel($N, E$)
 2:     $model \leftarrow SCHEMA$
 3:
 4:     $model \leftarrow$ “FACT TUPLE :”
 5:     for $\forall$ nodes in $N$ do
 6:         $model \leftarrow$ “$INSTANCE” + “ ” + nodeID + “ ” + nodeType
 7:     for $\forall$ edges in $E$ do
 8:         $model \leftarrow$ edgeType + “ ” + srcNodeID + “ ” + dstNodeID
 9:
10:     $model \leftarrow$ “FACT ATTRIBUTE :”
11:     for $\forall$ nodes in $N$ do
12:         $model \leftarrow$ nodeID + “{”
13:         for $\forall$ attributes in Node do
14:             $model \leftarrow$ attributeKey + “=” + attributeValue
15:         $model \leftarrow$ “}”
16:
17:     for $\forall$ edges in $E$ do
18:         $model \leftarrow$ ( + edgeType + “ ” + srcNodeID + “ ” + dstNodeID + ) + “{”
19:         for $\forall$ attributes in Edge do
20:             $model \leftarrow$ attributeKey + “=” + attributeValue
21:         $model \leftarrow$ “}”
22:
23:     Print model
```

simplicity of extending ClangEx.

Rex has two analysis modes: (i) full-analysis mode and (ii) simple-analysis mode , both of which analyze all source and header files similar to ClangEx's full-analysis mode. Rex's simple-analysis mode extracts information that pertains to ROS communications and the components that take part in these communications. This includes information about `classes` and the ROS `publishers`, `subscribers`, and `topics` contained inside them. Full-analysis mode builds upon simple-analysis mode by extracting information about ROS `publishers`, `subscribers`, and `topics` as well as information about basic C++ language features such as `classes`, `variables`, and `functions`. The choice of mode depends on the queries one might want to ask of the system. For instance, if one only wants to follow the dataflow between components, simple-analysis mode would be sufficient.

3.2.1 Rex Metamodel

Figure 3.6 shows the metamodel that describes models created using full-analysis mode of the Rex extractor. In this metamodel, there are two classes of nodes: C++ language features and ROS features. The top-level node type in Rex models is the `component` item. `Components` are sub-projects inside a main ROS project that separate the project into individual compilation units. The `compContains` relation connects `components` with any `class` located inside that component.

Because Rex is based upon ClangEx, many of the C++ language features it extracts conform to the same schema. Like in ClangEx, Rex maintains information about `classes`, `functions`, and `variables`. Relationships between these entities can include the entities and class might `contain` or function `calls`. Attributes are also recorded in ClangEx. As detecting potential feature interactions requires examining the control flow of a program, attributes such as `isControlFlow` is also recorded. This notes whether or not a variable participates in the decision portion of a control statement.

The `publisher` and `subscriber` nodes are ROS features that describe when a class communicates with some ROS topic. Publisher and topic entities are created when an `advertise` function call is made. This immediately creates a publisher entity and then forms an `advertise` relationship between the publisher and topic. Whenever a `publish` call is made, a unique relationship is established between that publisher and the topic. Whenever a `subscribe` call is encountered, a subscriber and topic entity is created and then a unique instance-specific relationship is created between the subscriber and topic. In addition, if that subscriber uses a callback function, a link is created between it and the callback function.

3.2.2 Advantages of the Rex Extractor

As Rex is based upon ClangEx, it shares many of ClangEx's advantages including its use of the Clang API and its ability to analyze a subset of a project. The notable advantage of Rex is that it is able to capture messages sent between components in a distributed software system. This allows users to write relational queries that can detect interactions between the components in a distributed system. Furthermore, although Rex targets only ROS-based projects, insights that were gained from building Rex could be reused or adapted in the development of new extractors for different messaging frameworks (e.g. components connected by a bus like CAN or a network connection like Ethernet).

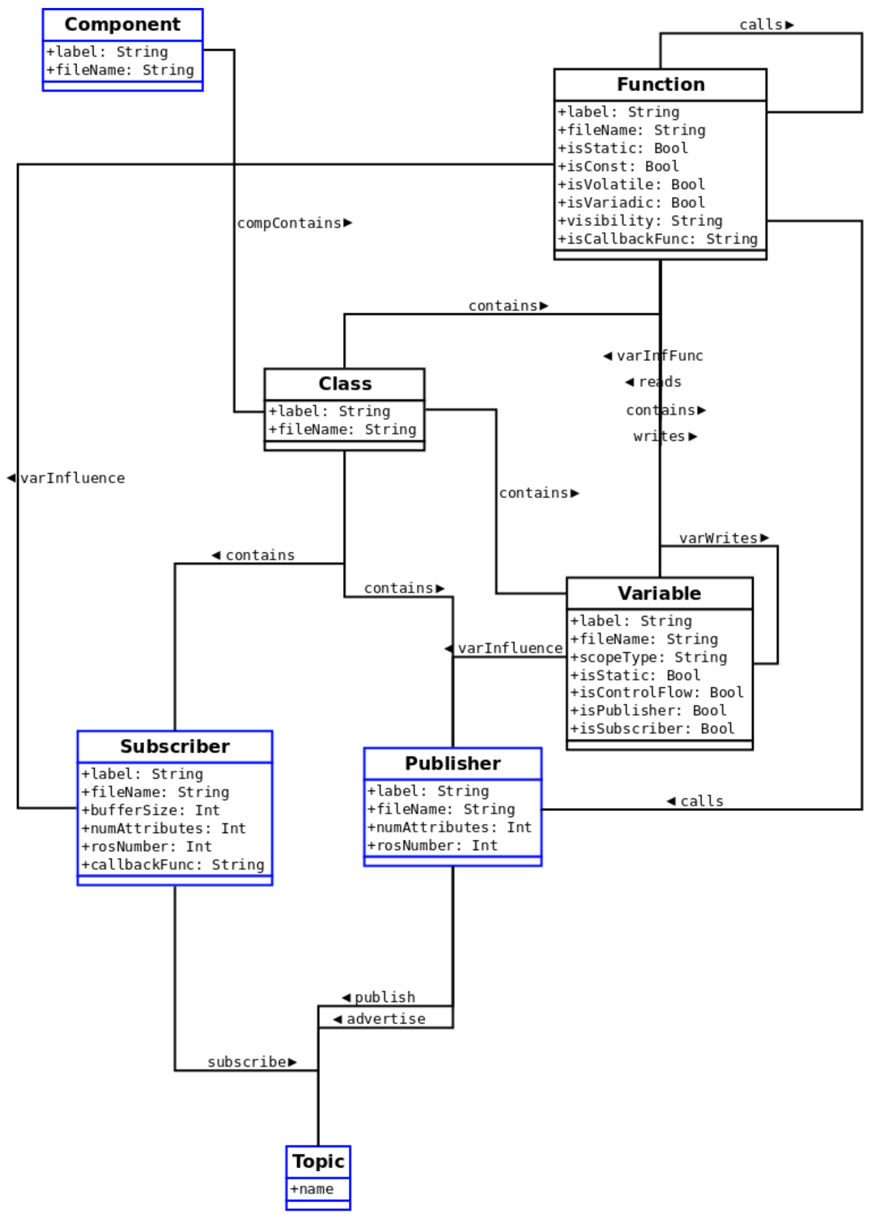


Figure 3.6: The Rex metamodel for full-analysis mode. Blue classes are ROS features and black classes are C++ features.

3.2.3 Disadvantages of the Rex Extractor

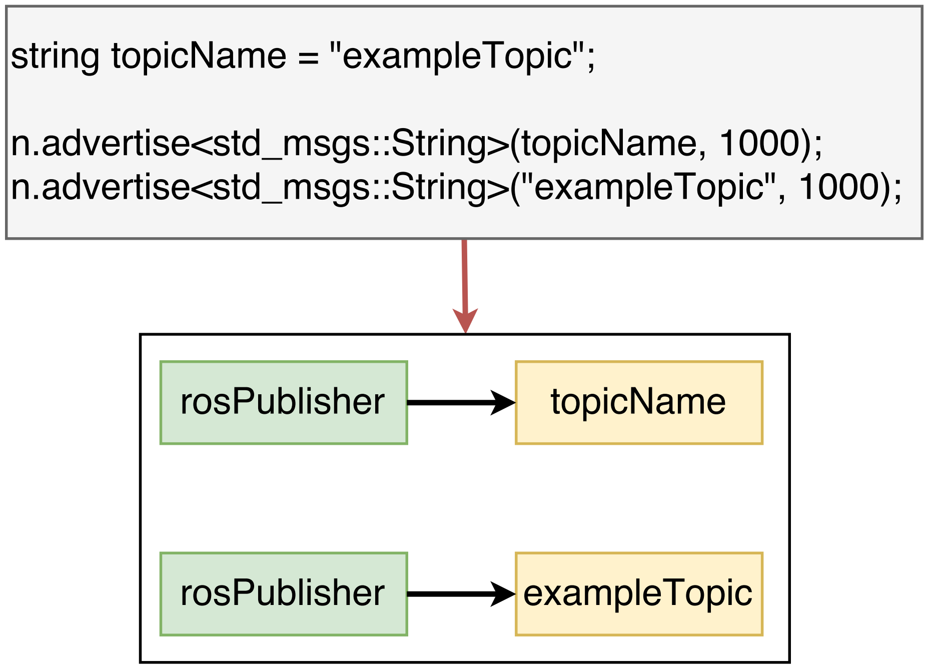
As Rex is built on top of ClangEx, many of the disadvantages of the ClangEx extractor are also present in Rex. This includes being bound to the Clang compiler and requiring the use of compilation databases. In addition, Rex has several disadvantages related to how it extracts ROS projects.

Rex has the potential of generating erroneous models due to static-analysis limitations. It is also possible for ClangEx to generate incorrect facts, but Rex suffers more from this issue due to how ROS topics are created. Rex extracts ROS topics by looking at function calls to the `advertise` and `subscribe` ROS library functions, which means that Rex can be fooled into adding incorrect topics or relationships into the final model. It is expected that components publish or subscribe to a topic and reference topic names using string **literals**. If instead a component refers to a topic through a string **variable**, then Rex will interpret the string literal and string variable as two different topics even if the two strings are equivalent. Figure 3.7 shows this limitation in detail. Thus, before Rex can be run, the publisher and subscriber code must be checked to ensure that string literals are being used for the topic-name parameter in `advertise` and `subscribe` calls. Publish calls do not take a topic-name parameter because they already point to a topic when created through `advertise`.

3.2.4 Rex Internals

The implementation of Rex is very similar to that of ClangEx. Rex uses Clang's LibTooling API and AST Walkers to analyze parsed C++ source code. The major difference is that Rex uses Clang's AST visitor API rather than AST matchers to gather information about the source code. Note that, because Clang does not have AST matchers that detect specific ROS code fragments like `advertise` or `subscribe` calls, using AST matchers would still require Rex to narrow down these matches to select only `advertise` and `subscribe` calls. Thus, Rex has its own checks to look for particular situations such as calls to `publish` or `subscribe` to ROS topics. Whereas AST matchers would pass control over to ClangEx if a certain AST situation was met, Clang's AST visitor API allows for Rex to run code every time a type of AST node is encountered.

Other than these differences, Rex uses similar modules to ClangEx in a similar manner. Like ClangEx, there are three modules: 1. the Driver module that carries out command line operations and activates Clang, 2. AST Walker modules that carry out how the AST is processed, and 3. the Graph module that maintains an in-memory graph of the source



```
string topicName = "exampleTopic";

n.advertise<std_msgs::String>(topicName, 1000);
n.advertise<std_msgs::String>("exampleTopic", 1000);
```

Figure 3.7: A situation with topic names where Rex generates an invalid model.

code being analyzed. As Rex maintains similar modules compared to ClangEx, Figure 3.2 (from Section 3.1) shows how the modules in Rex interact.

Due to the similarity between Rex and ClangEx, instead of providing a detailed description of the modules in Rex, the remainder of this section will describe features in the Walker and Graph modules that are unique to Rex. This includes ROS specific detectors and an improved variable read/write detection system.

AST Walker Modules

The AST Walker module operates in a similar fashion to the ClangEx AST Walker module with some major differences. Instead of using the AST matcher API to match certain language features in the AST, Rex runs code each time any high-level declaration or statement is encountered. This means that Rex code executes whenever a `function`, `variable`, or `class` declaration is encountered or any time a C++ statement is detected. When Rex receives an AST node, it checks the node to ensure it has not been encountered before. If it has not, the AST Walker module then will determine whether that AST node represents a declaration or statement of a specific type. For instance, declarations of interest are `variables`, `classes`, or `functions` and statements of interest are call expressions, binary/unary expressions, constructor call expressions, and decision points of control structures. For each of these subtypes, Rex has specific code that runs to add the corresponding node(s) and edge(s) to the hierarchical graph in the Graph module. For example, detecting a call-expression statement between two functions would cause a `calls` relation to be added to the in-memory graph.

The Rex AST Walker also has some special features for it to generate a model for detecting feature-interaction hotspots. As mentioned above, Clang does not have built-in detectors that are capable of finding ROS language features like publishers or subscribers. As such, the AST Walker has custom `publisher` and `subscriber` detectors that look for function calls originating from a `NodeHandle` object. If a `ros::subscribe` or `ros::advertise` call is encountered, Rex creates an associated subscriber or publisher node. Rex has to remember the last publisher created and then look for the variable that is going to store that publisher object so that when a subsequent publish call is made, Rex knows which topic that publisher is recording data to. Rex has to remember the last publisher created because the it requires multiple AST nodes to create an object and assign it to a variable.

Creating `topic` nodes is a little more complicated because topics are not created through their own function calls. When a publisher or subscriber is created, one of the parameters is expected to be a string literal that is the name of the topic that the publisher or subscriber points to. Therefore, when Rex detects a function call that creates a publisher and subscriber, Rex examines the parameters to look for the parameter value that names the topic. It then creates a topic node in the graph with the parameter obtained from the advertise/subscribe function call. If this parameter is a variable rather than a string literal, then Rex cannot resolve the topic name because variable values are not known at compile time. Instead, Rex will just use the variable name. That means that if two advertise calls use variables to specify to topic and the variable names are different (even if the topic name contained in the variables are the same), Rex will create two different topic nodes for the same topic.

The AST walker detects whether a function reads or writes to a variable. This information is useful as it allows users to write Grok queries that can trace the flow of data between components. For instance, a user could write a Grok query that determines all the variables that a ROS topic modifies. Algorithm 2 shows the general algorithm that detects variable accesses from a C or C++ statement. The algorithm shown here is a scaled down version of the one that is in Rex and is just for illustration. Because expressions can be composed of numerous sub-expressions, this algorithm starts with a top-level expression and decomposes into each sub-expression recursively.

Algorithm 2 Basic Read/Write Detection Algorithm

```
 1: procedure readWriteDetection($expr, varMap, upperVal$)
 2:     if $expr$ isa $BinaryExpression$ then
 3:         $operator \leftarrow expr.$getOperator()
 4:         $lhsExpr \leftarrow expr.$getLHS()
 5:         $rhsExpr \leftarrow expr.$getRHS()
 6:
 7:         $currentVal \leftarrow$ getReadOrWriteForLHS($upperVal, operator$)
 8:
 9:         if $lhsExpr$ isa $DeclRef$ then
10:             $varMap.$add($lhsExpr, currentVal$)
11:             return $varMap$
12:         else
13:             $varMap \leftarrow varMap +$ readWriteDetection($lhsExpr, varMap, currentVal$)
14:
15:         $currentVal \leftarrow$ getReadOrWriteForRHS($upperVal, operator$)
16:
17:         if $rhsExpr$ isa $DeclRef$ then
18:             $varMap.$add($rhsExpr, currentVal$)
19:             return $varMap$
20:         else
21:             $varMap \leftarrow varMap +$ readWriteDetection($lhsExpr, varMap, currentVal$)
22:         return $varMap$
23:
24:     else if $expr$ isa $UnaryExpression$ then
25:         $operator \leftarrow expr.$getOperator()
26:         $baseExpr \leftarrow expr.$getBase()
27:
28:         $currentVal \leftarrow$ getReadOrWriteUnary($upperVal, operator$)
29:
30:         if $baseExpr$ isa $DeclRef$ then
31:             $varMap.$add($baseExpr, currentVal$)
32:             return $varMap$
33:         else
34:             return readWriteDetection($baseExpr, varMap, currentVal$)
```

The algorithm is invoked by passing in the target expression, an empty hash map where the keys are variables and values are access types (either `Read`, `Write`, or `Both`), and an upper access type (which when invoked should be `null`). Then, the algorithm determines the type of expression (binary, unary, or a variable[3]). If the expression is binary then the algorithm gets the left-hand and right-hand sub-expressions (labeled LHS and RHS respectively) as well as the central operator. It then uses two functions called `getReadOrWriteForLHS` and `getReadOrWriteForRHS` which get whether the LHS and RHS are being read or written to based on the upper access type and the central operator in the binary expression. Then, for both the LHS and RHS, if the sub-expression is a variable, it simply adds that variable and the access type to the $varMap$ hash map. If the sub-expression is another binary or unary expression, the algorithm calls itself with the LHS or RHS sub-expression, the current $varMap$ and the current access type contained inside $currentVal$ (to propagate it downwards). If the expression is a unary expression, the algorithm just gets the operator of the unary expression and the base expression. Then, it uses a function called `getReadOrWriteForUnary` to get the access type for that expression based on the $upperVal$ and current operator. Once the access type is determined, the algorithm then checks if the sub-expression is a variable or another expression. If its a variable, it simply adds the access type contained within $currentVal$ to the $varMap$ for that variable. If it is an expression, the algorithm recursively calls itself with the sub-expression, $varMap$, and current access type contained in $currentVal$.

Figure 3.8 shows an example of the read/write detection algorithm in action. This figure determines the variable access types contained inside the basic C++ expression $variable1 = (variable2 += variable1)$. As per the AST, this expression is a binary expression centered around the = sign with the left-hand side (LHS) being $variable1$ and right-hand side being $(variable2 += variable1)$. According to Algorithm 2, the LHS is written to and the RHS is read from. Then, Algorithm 2 is recursively called for both the LHS and RHS of that expression. When Algorithm 2 is invoked on the LHS, it records that in the parent expression the LHS side was written to and notes that the expression $variable1$ is just a declaration reference expression. Because a declaration reference is a reference to a `variable`, the algorithm records that $variable1$ was written to.

On the RHS of $variable1 = (variable2 += variable1)$, the algorithm records that in the parent expression, the RHS side was read from. Here, as $(variable2 += variable1)$ is surrounded by parentheses, the algorithm strips away the parentheses and then notes that this expression is binary centered around the += operator with the LHS being $variable2$

[3]This is extremely simplified to reduce the complexity of this algorithm. As per the C++ AST, expressions could also be parenthesis expressions, block expressions, or others. This is addressed in the full algorithm contained in Rex.

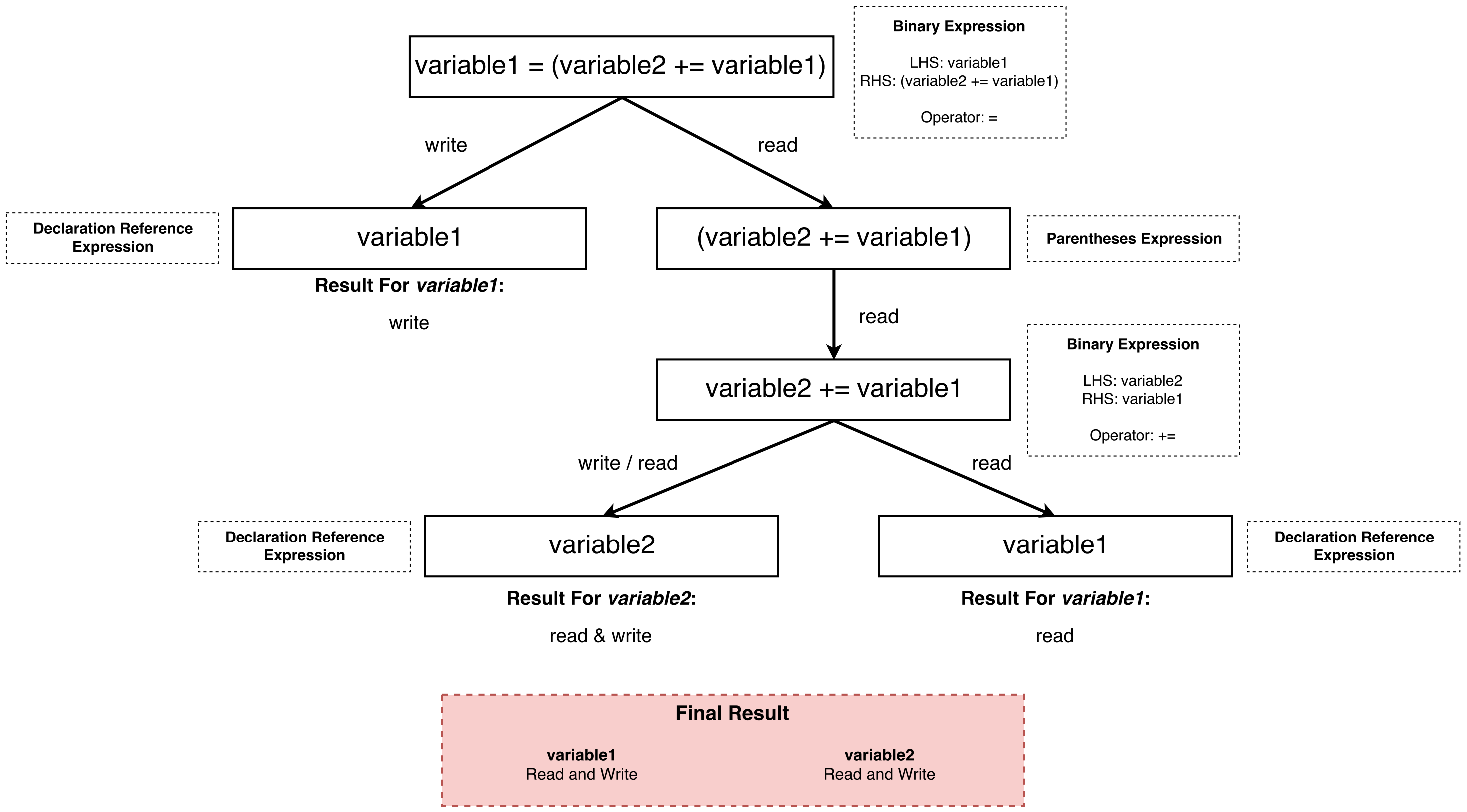


Figure 3.8: An example of the variable read/write detection algorithm.

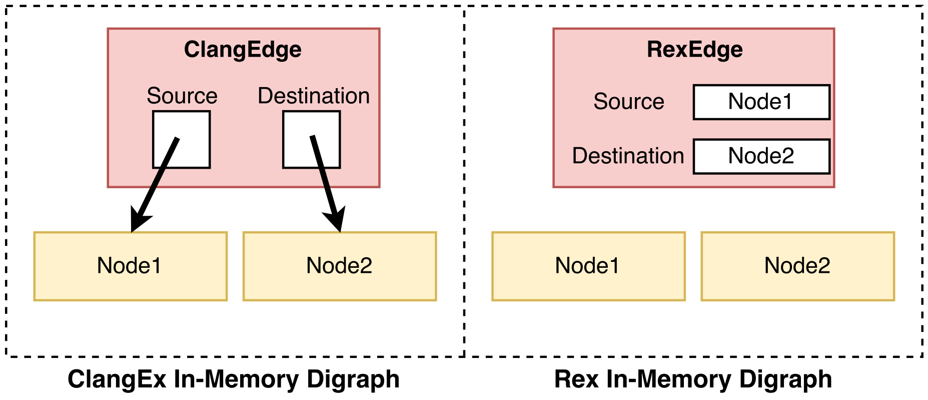


Figure 3.9: A comparison of how graphs in ClangEx and Rex are stored.

and RHS being $variable1$. Because the $+=$ compound operator is involved, the LHS is read from and written to and the RHS is just read from. As such, algorithm 2, recursively calls itself for the LHS and RHS passing down the variable access values (write and read for the LHS and read for the RHS). As both the LHS and RHS expressions are just variables, the algorithm records the access types for both variables. As such, for the expression $variable1 = (variable2 += variable1)$, both $variable1$ and $variable2$ are found to be read from and written to.

Graph Module

Rex's Graph module is an improved version of the ClangEx Graph module. It maintains a digraph that stores a collection of nodes and edges. Nodes are called `RexNodes` and can be of several different types that reflect language and ROS features. Edges are called `RexEdges` and have a source RexNode and a destination RexNode that participate in the relationship. Edges also have different types that symbolize different relationships that can occur between the nodes. The names of the edge types follow the metamodel shown in Figure 3.6.

Compared to ClangEx, Rex manages hierarchical graphs slightly differently. The major difference is in how each deal with undefined references during edge creation. As previously stated, when analyzing source code, some edges may be encountered prior to encountering the declaration of source or destination nodes to which the edge refers. As a result, there needs to be a way to create and describe these edges prior to the creation of the nodes. Instead of having a list of undefined edges that have to be resolved after all artifacts are analyzed (as ClangEx), RexEdges just store the ID of the source and destination node. Figure 3.9 shows an example of how Rex edges and ClangEx edges are represented in the hierarchical graph: in ClangEx edges use pointers to refer to nodes that participate in relations, and in Rex edges use strings to store unique IDs of nodes that participate in relations. For Rex, before an edge is written out in TA format, it is checked to ensure both the source and destination node exist. The advantage of this approach is that multiple structures do not have to be used to keep track of these undefined references and code complexity is reduced.

3.3 Chapter Summary

In this chapter we presented two different fact extractors for analyzing C and C++ source code: ClangEx and Rex. ClangEx is designed to extract general language facts from source code whereas Rex captures information about the communication of different features in a ROS-based system. We note that Rex is built upon ClangEx and that both use similar extraction methodologies. Finally, the Rex extractor is used in the case study presented in Chapter 5 as it is capable of extracting information that allows for the detection of potential feature interactions.

Chapter 4

Hotspots in Automotive Systems

As software continues to play an increasing role in the automotive domain, it is important for developers to identify potential feature interactions. Due to its generality and extensibility, the relational algebra toolchain is a good fit for detecting these potential feature interactions. The major difference between analyzing monolithic systems and automotive systems using this toolchain is that automotive systems rely on message-passing for features to communicate. Further, these features are often distributed across a plethora of heterogeneous computing cores. Each feature has the potential of sending messages to other features than can affect behaviour. As a result, a fact extractor and Grok scripts need to be developed so that this message passing information is appropriately captured and analyzed.

There are three different classes of hotspots that might indicate potential feature interactions in an automotive systems. These three hotspot classes are not exhaustive and might depend on the vehicle's communication architecture. As before, the goal of these hotspots are to identify areas that should be investigated further using other tools. The three hotspot categories are *feature-communication* hotspots, *multiple-input* hotspots, and *control-flow* hotspots. Figure 4.1 provides a breakdown of each hotspot class each which contain several different hotspots that each identify a specific issue in the software project. Feature-communication hotspots relate to the flow of information among the features in the project. Multiple-input hotspots identify instances where multiple features might send messages to the same feature near simultaneously which could result in a race condition. Lastly, control-flow hotspots identify instances where a communication flow between features might affect the behaviour of the destination feature. Each of these hotspot types are related to one another; instances of one hotspot might also be considered an instance of another hotspot. Figure 4.2 shows a set diagram that highlights the relationships between each hotspot type. Note that the inter-component-based communication hotspot contains all other hotspot types and that the behaviour-alteration and publish-alteration hotspots detect similar situations.

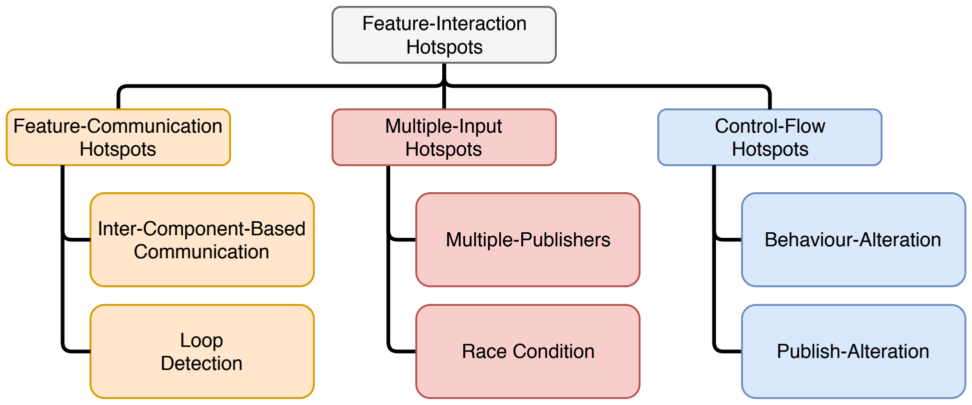


Figure 4.1: The breakdown of feature interaction hotspots.

The remainder of this chapter introduces each class of hotspots and gives a definition of the hotspots that are part of each class. With each hotspot definition, a Grok script is presented to show the steps required to detect that hotspot in Grok. As these scripts are simplified, Appendix C presents all hotspot scripts used in the case study and provide a detailed description of how they operate. For each hotspot, it is assumed that a model exists with a collection of entities, relations, and attributes. Table 4.1 gives each of the entities, relations, and attributes used in these scripts and describes their format and usage.

The remainder of this section is as follows: Section 4.1 describes feature-communication hotspots. Section 4.2 describes multiple-input hotspots. Section 4.3 describes control-flow hotspots.

4.1 Feature-Communication Hotspots

Feature-communication hotspots highlight areas in the automotive software system where one feature communicates with another. This includes direct and indirect communications between a source and destination feature or loops in the communication graph. There is inherent value in detecting these hotspots as they allow project stakeholders to gain a better understanding of how the features in the system interact. Since automotive projects tend to be spread across numerous teams [50], a developer might not know precisely the

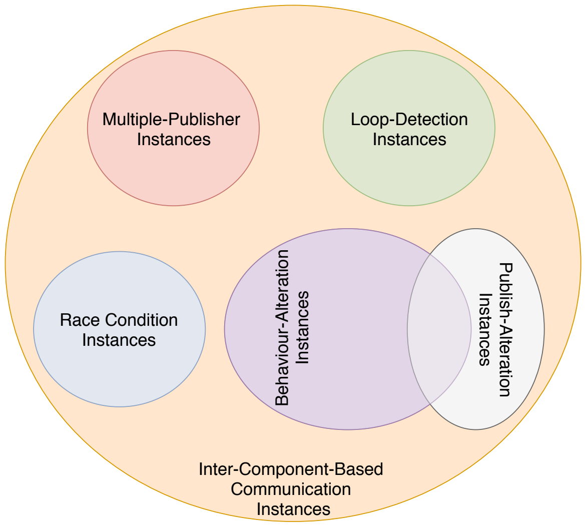

Multiple-Publisher Instances
Loop-Detection Instances
Race Condition Instances
Behaviour-Alteration Instances
Publish-Alteration Instances
Inter-Component-Based Communication Instances

Figure 4.2: A set diagram showing the relationships between each hotspot type.

Relation/Attribute	Description
`direct <Function1> <Function2>`	`<Function1>` in one feature sends a message to `<Function2>` in another feature.
`call <Function1> <Function2>`	`<Function1>` calls `<Function2>`.
`write <Function> <Variable>`	`<Function>` assigns data to `<Variable>`.
`varAssign <Variable1> <Variable2>`	`<Variable1>` assigns its data to `<Variable2>`.
`varInfFunc <Variable> <Function>`	`<Variable>` is used in a control flow statement and affects whether `<Function>` is called.
`varInfluence <Variable> <Function>`	`<Variable>` is used in a control flow statement and affects whether `<Function>`, in another feature, receives data.
`<Function> { receiveFunc = <true/false> }`	Denotes whether `<Function>` receives data from another feature.
`<Function> { sendFunc = <true/false> }`	Denotes whether `<Function>` sends data to another feature.
`<Variable> { isControlFlow = <true/false> }`	Denotes whether `<Variable>` is in the decision condition of a control flow statement.

Table 4.1: Relations and attributes that are used in the Grok scripts in this chapter.

other features with which their feature communicates indirectly, or might not be aware that their feature communicates with itself indirectly.

Hotspot Name	*Description*
Inter-Component-Based Communication (ICBC)	Detects features that communicate with other features directly or indirectly. For indirect messages,it detects a sequence of inter-feature communications.
Loop Detection (LD)	Detects communication loops where a feature communicate with itself directly or indirectly. Uses the inter-component-based methodology to detect communications.

Table 4.2: The two hotspots part of the feature-communication class.

There are two main hotspots in this class: *inter-component-based communication* and *loop detection*. Table 4.2 provides an overview for both hotspot. The remainder of this section describes both in detail.

4.1.1 Inter-Component-Based Communication

The goal of the inter-component-based communication hotspot is to detect cases of direct and indirect inter-feature communications. This means that, for an instance of this hotspot to be detected, there has to be a continuous path of function calls (where some are publish/callback functions associated with message passing, and some are ordinary functions) from start feature to end feature.

Figure 4.3 shows an example of a direct and indirect case of the inter-component-based communication hotspot. In the direct case, through the `sendMessage` function, feature `A` sends a direct message to feature `B` that is received by the `receiveMessage` function. In the indirect case, through the `sendMessageA` function, feature `A` sends a message to feature `x` which is received by the `receiveMessageX` function. Then, after an unspecified number of function calls inside feature `x`, the `sendX` function sends a message to another feature and so on; the sequence ends with a function inside a feature that sends a message received by the `receiveMessageB` function in feature `B`. Importantly, this chain of communications does not mean that one feature receiving a message causes another message to be sent out. Rather, what is reported is a **possible** sequence of communications.

The major advantage of this hotspot is that it provides detailed results to how features communicate. However, by detecting communications at the functional level, it may also result in false negatives. For instance, consider Figure 4.3 except whenever feature `A` sends a message to feature `B`, feature `B` changes state. Then, whenever `B` changes state, it sends a message to feature `C`. In this situation, although a communication from `A` to `C` *technically* does not take place, feature `A` is still causing feature `C` to receive a message.

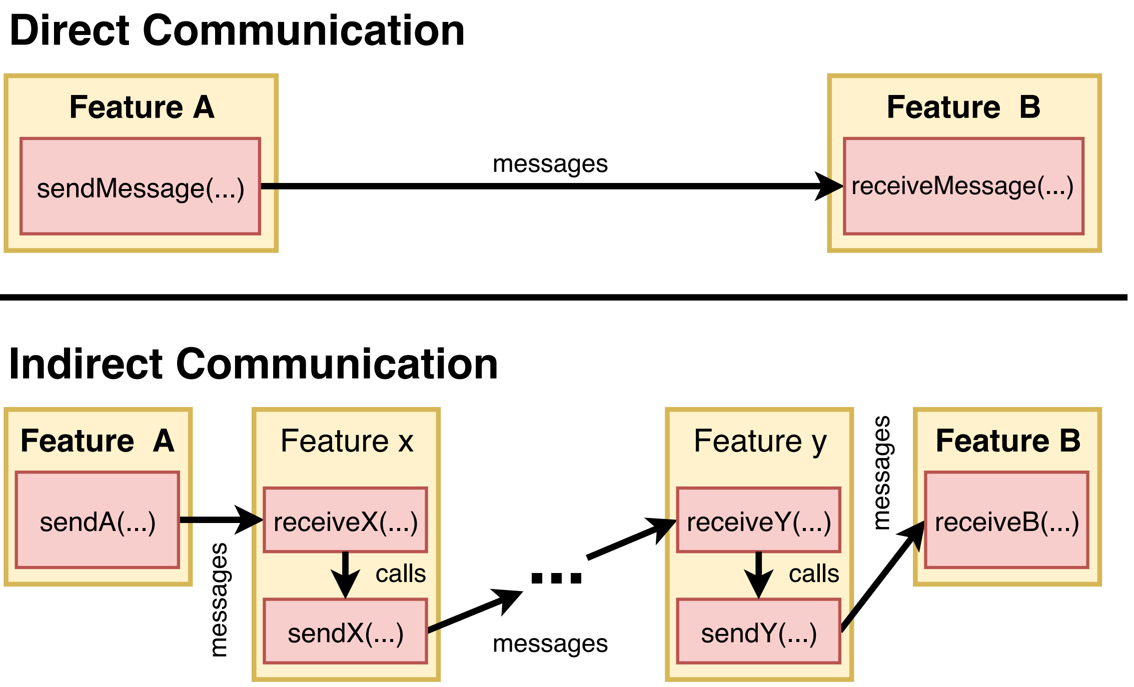


Figure 4.3: The difference between direct and indirect communications between features `A` and `B`.

Detecting this hotspot requires a small model that has only a few entities and relations: (1) feature entities, (2) function entities, (3) a relation called `direct` that describes which features directly communicate with other features, (4) a relation called `call` that describes which functions call other function, (5) an attribute called `receiveFunc` which describes functions that receive messages from other features, and (6) an attribute called `sendFunc` which describes functions that send messages to other features. With such a model, Figure 4.4 shows the Grok operations to detect inter-component-based communication hotspots. This hotspot can be detected using the following steps:

```
1 direct;
2
3 fullCall = direct + call;
4 fullCall = fullCall+;
5
6 fullCall = (sendFunc . {true}) o fullCall o (receiveFunc . {true});
7
8 indirect = indirect - direct;
9 indirect;
```

Figure 4.4: The Grok syntax for detecting the inter-component-based communication hotspot.

1. For direct messages simply print the contents of the `direct` relation. This is illustrated in line 1 of Figure 4.4.

2. Using the union operator, combine the `direct` and `call` relations together. This creates a relation with function calls and feature messages. Call this set `fullCall`. Then, compute the transitive closure of `fullCall`. Lines 3 and 4 in Figure 4.4 shows the Grok syntax to do this.

3. Remove from `fullCall` any entry whose domain or range element does not start with the last function in a feature and end with the first function in a feature respectively. This ensures that the resulting relation represents an execution path that starts with a publish call and ends with a message being received by a subscriber. Store these results in a relation called **indirect**. Line 6 shows the Grok syntax to do this.

4. Remove any entries from `indirect` that are also in `direct` and store the results back in `indirect`. Print the results. This occurs in lines 8 and 9.

4.1.2 Loop Detection

Loop detection is the second feature-communication hotspot. Its purpose is to detect communication loops that are either self-loops (direct) or a loop consisting of several features (indirect). A loop can be defined as a sequence of communications from a feature back to itself. Loops can be detected by taking the component-based communication instances or inter-component-based communications and looking for loops in the communication graph.

This hotspot is an extension of the inter-component-based communication hotspot. Similar to the TA model needed for the inter-component-based communication hotspot

```
1  featureRel = id feature;
2
3  directLoop = direct ^ featureRel;
4  directLoop;
5
6  fullCall = direct + call;
7  fullCall = fullCall+;
8
9  indirectLoop = (sendFunc . {true}) o fullCall o (receiveFunc . {true});
10 indirectLoop = indirectLoop ^ featureRel;
11 indirectLoop;
```

Figure 4.5: The Grok syntax for detecting the loop detection hotspot.

detection, the TA model needs to contain the following facts: (1) feature entities, (2) function entities, (3) a relation called `direct` that describes which features directly communicate with other features, (4) a relation called `call` that describes which functions call other functions, (5) an attribute called `receiveFunc` which describes functions that receive messages from other features, and (6) an attribute called `sendFunc` which describes functions that send messages to other features. With this model, Figure 4.5 shows the Grok operations to detect loops. This hotspot can be detected using the following steps:

1. Generate a relation of features where, for each feature in the model, the domain and range is the feature ID. This is achieved in line 1 of of Figure 4.5 by taking the identity of the set of features.

2. For direct loops, take the `direct` relation and remove any entry whose domain and range elements are not the same. Print the results. This occurs in lines 3 and 4 of the Grok script.

3. For indirect loops, combine the `direct` and `call` relations together using the union operator. This creates a relation with function calls and feature messages. Call this set `fullCall`. This occurs in line 6 of the Grok script.

4. Compute the transitive closure of `fullCall` (line 7 of the Grok script) and remove any entry whose domain and range elements do not start with the sending function in a feature and end with the receiving function in a feature respectively. This occurs by joining any last functions with the domain of the `fullCall` relation and any first functions with the range of the `fullCall` relation. Line 9 shows the syntax to do this. Store these results in the `indirectLoop` relation. Remove from this relation

any entry whose domain and range elements are not the same. This is done through intersection as shown in line 10.

5. Print the `indirect` relation to get the indirect inter-component-based communication loops. Line 11 of the Grok script prints the results.

4.2 Multiple-Input Hotspots

The multiple-input hotspot class detects situations where multiple features communicate with the same recipient feature. Such a scenario might indicate a bug where a message from one feature is superseded (possibly repeatedly) by messages from another competing feature. This hotspot class can also detect race-condition situations where data received from multiple features update a single variable. In this case, updates from one of the competing features could potentially be lost.

Hotspot Name	*Description*
Multiple Publishers (MP)	Detects if multiple features can communicate with the same feature through the same callback function.
Race Condition (RC)	Detects if multiple features can communicate with the same feature causing competing updates to the same variable.

Table 4.3: The two hotspots part of the multiple-input class.

There are two hotspots in this class: the *multiple-publishers* hotspot and the *race condition* hotspot. Table 4.3 provides an overview of the two hotspots in this class. The remainder of this section presents the two hotspots in detail and describes how they can be detected using relational algebra.

4.2.1 Multiple-Publishers

The multiple-publishers hotspot detects if multiple features can communicate with the same function inside a feature. Any method to detect this hotspot needs to take this communication architecture into account. For instance, in ROS, features publish data to topics which are then sent to features that subscribe to these topics. Therefore, to detect this hotspot in ROS, we need to detect whether: (1) multiple publishers send data to the same topic, and (2) multiple topics send data to the same callback function.

This hotspot can uncover the possibility that multiple features send data to a single recipient function causing one feature to overpower the other features. In an architecture like ROS, this is an important hotspot to detect because each feature has an incoming and outgoing message buffer for each topic it subscribes and publishes to. If one feature floods this queue, messages from other features are dropped.

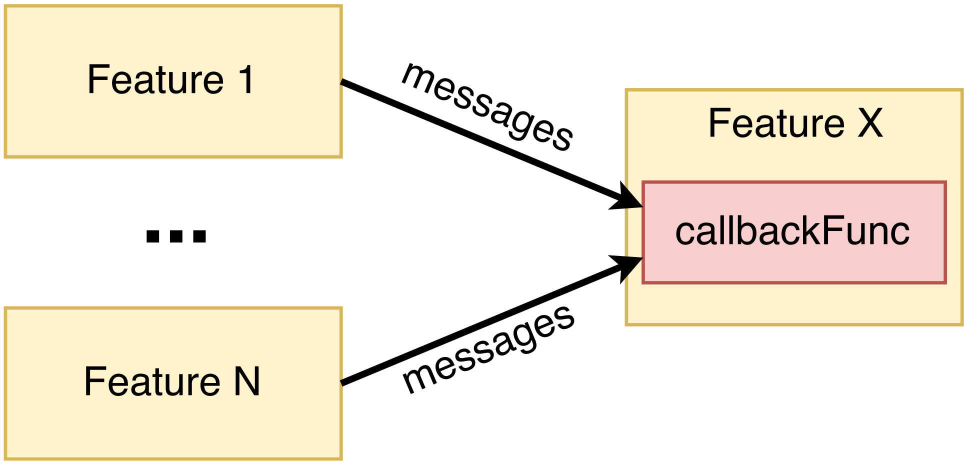


Figure 4.6: An example of the multiple-publisher hotspot.

Figure 4.6 shows an example of the multiple-publishers hotspot. In this figure there are a collection of features shown in yellow and a single recipient function called `callbackFunc` shown in red. Features 1 through N send messages to feature X through the `callbackFunc` function.

Detecting this hotspot using relational algebra requires a simple TA model that has the following elements: (1) feature entities, (2) function entities, and (3) a relation called `direct` that describes which features directly communicate with other features. The model used to detect this hotspot does not have to be detailed because this hotspot is only concerned with calls between features. Figure 4.7 shows the Grok operations to detect multiple-publisher hotspots. This hotspot can be detected using the following steps:

1. Get the cardinality of the range of the `direct` relation. Store it in a relation called `cardDirect`. This is done by using Grok's built-in `indegree(...)` function as shown in line 1 of Figure 4.7.

```
1 cardDirect = indegree(direct);
2 cardDirect = cardDirect [ &1 > 1 ];
3
4 direct = direct o dom cardDirect;
5 direct;
```

Figure 4.7: The Grok syntax for detecting the multiple-publishers hotspot.

2. Remove any entries from the `cardDirect` relation if the cardinality is 1 or fewer. This can be done by using Grok's selection operator on the range of the `cardDirect` relation as shown in line 2.

3. For the remaining entries in `cardDirect`, take the domain of the relation and join it with the range of `direct`. This gets all publications to the same callback functions in the project. This is shown in line 4 of the Grok script. Then, as per line 5, print the results of `direct`.

4.2.2 Race Condition

The race condition hotspot detects if multiple features send messages that end up modifying the same variable. Because a feature can receive messages from a variety of different features through different callback functions, there is the potential for these callback functions to each write to the same global variables. Figure 4.8 shows an example of this type of hotspot. Here, two features called A and B each message a separate callback function inside feature C at some point in time. Then, each of these callback functions write to a variable called `raceVar`.

Detecting this hotspot is important because it has the potential of identifying race conditions within a feature.

Detecting race condition hotspots using relational algebra requires a model that is a little more detailed compared to the module used for detecting multiple-publishers hotspots because detecting this hotspot also requires knowledge of variables being written to. In the following, bold text indicates the differences between this model and the multiple-publisher hotspot model. As such, the following is required: (1) feature entities, (2) function entities, (3) **variable entities**, (4) a relation called `direct` that describes which features directly communicate with other features, (5) **a relation called `write` describing which functions write to which variables**, and (6) **an attribute called `receiveFunc` which describes functions that receive messages from other features**. Figure 4.9 shows

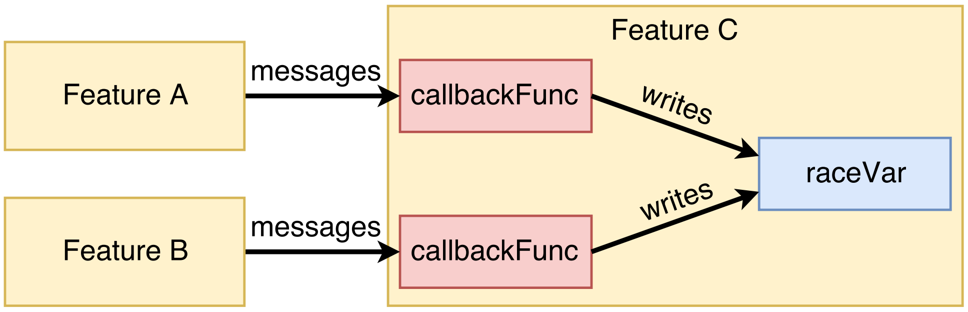


Figure 4.8: An example of the race condition hotspot.

the Grok operations to detect race condition hotspots. This hotspot can be detected using the following steps:

1. Get the callback functions by taking the `receiveFunc` attribute and selecting entries that are true. This gets all functions that receive data. This is denoted in line 1 of Figure 4.9. Store the results in `callbackFunc`.

2. For the `write` relation, keep only entries involving a callback function writing to a variable. Store this in a new relation called `callbackWrite`. The relation captures all the variables that each callback function writes to. This is represented in line 3 of the Grok script.

3. Get the cardinality of the `callbackWrite` relation using the `indegree(...)` function. This gets the cardinality of the variables being written to. Then, use the selection operator to keep only entries where the cardinality is greater than 1. Lines 5 and 6 in the Grok script show how this is achieved.

4. Join the original `callbackWrite` relation with the domain of the `cardVars` relation to keep only entries where multiple callbacks write to the same variable. Store this join back in `callbackWrite`. Print the results. Lines 8 and 9 show the Grok syntax to do this.

4.3 Control-Flow Hotspots

Another class of hotspots that can be problematic are control-flow hotspots. These hotspots identify situations where the behaviour of one feature is altered due to messages received

```
callbackFunc = receiveFunc o {true};

callbackWrite = callbackFunc o write;

cardVars = indegree(callbackWrite);
cardVars = cardVars [ &1 > 1 ];

callbackWrite = callbackWrite o dom cardVars;
callbackWrite;
```

Figure 4.9: The Grok syntax for detecting the race condition hotspot.

from other features. For instance, one feature sending a message to another feature causing it to change state is an example of a control-flow hotspot. These state changes could also lead to a feature sending messages to other features. These hotspots identify these parts of the software where interactions between features have a definitive effect on one another's behaviour which, if not intended, warrants further investigation.

Hotspot Name	*Description*
Behaviour Alteration (BA)	Detects if a received message (from another feature) can cause a feature to alter how it behaves.
Publish Alteration (PA)	Detects if a received message (from another feature) can cause a feature to send messages to other features.

Table 4.4: The two hotspots part of the control-flow class.

There are two major hotspots in this class: the *behaviour-alteration* hotspot and the *publish-alteration* hotspot. Table 4.4 provides an overview of these two hotspots and how they relate to other hotspots. The remainder of this section presents these two hotspots and describes how they can be detected using relational algebra.

4.3.1 Behaviour-Alteration

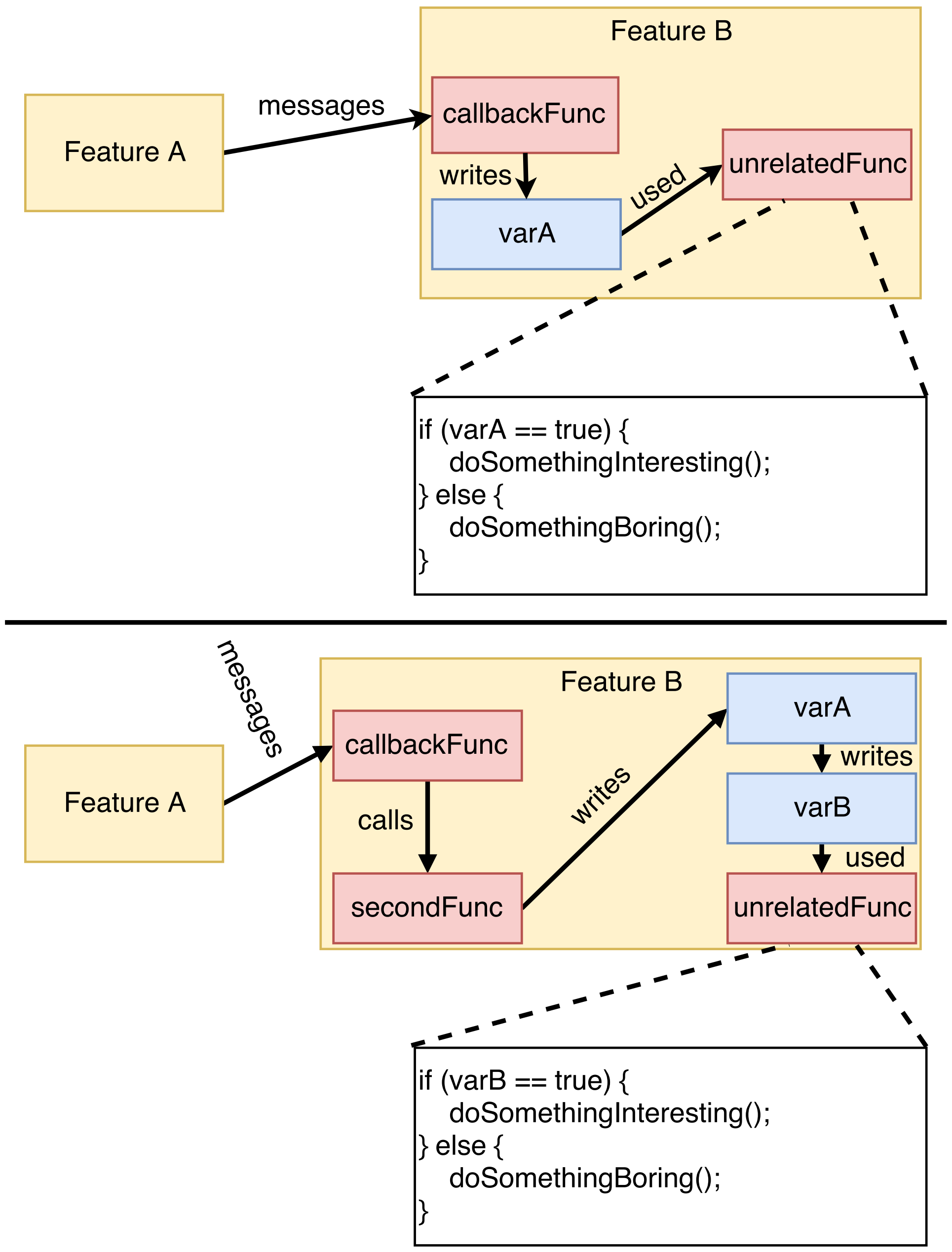
The behaviour-alteration hotspot detects if a message from one feature causes another feature to change its behaviour. The behaviour of the second feature is considered changed if the message callback function writes to a variable (calls a function called directly or indirectly that writes to a variable), and this variable is eventually used in the decision condition of a control structure such as a `for` loop or `if` statement. The variable that is used in the decision block does not have to be the variable originally written to; variables can be written to by the callback function and then be used in assignments to other variables, which are used in control structures.

Detecting this hotspot is important because it provides even greater detail than feature-communication hotspots, in that it reveals how information received from other features can change the behaviour of a feature. With the results from this hotspot, engineers can place further attention on the code leading to the changed behaviour to ensure that it is desired.

Figure 4.10 shows two different instances of the behaviour-alteration hotspot. These two instances are not exhaustive because there are numerous ways in which this hotspot manifests itself. As before, yellow boxes are features, red boxes are functions, and blue boxes are variables. In the first configuration, feature A communicates with B causing a variable called `varA` to be modified. This variable is then used in an `if` statement. From both these examples, feature B's behaviour is being modified as a result of a message from feature A. In the second example, feature A publishes a message received by the `callbackFunc` function in feature B which then calls the `secondFunc` function. Inside `secondFunc`, `varA` is written to. Then, that variable is used in an assignment to another variable called `varB`. Lastly, `varB` is used in an `if` statement inside `unrelatedFunc`.

Detecting this hotspot using relational algebra requires a fairly detailed model that includes: (1) feature entities, (2) function entities, (3) variable entities, (4) a relation called `direct` that describes which features directly communicate with other features, (5) a relation called `call` that describes which functions call which other functions, (6) a relation called `write` that describes which functions write to which variables, (7) a relation called `varAssign` that denotes which variables assign their values to other variables (the source of this relation is the assigner and the destination is the assignee), (8) a relation called `varInfFunc` of the form `<VARIABLE> <FUNCTION>` that for variables that participate in the decision condition of a control structure, highlights any function calls that are nested within, (9) an attribute called `isControlVar` which represents whether a variable is used in the decision condition of a control-flow structure, and (10) an attribute called `receiveFunc` which describes functions that receive messages from other features. Given such a model exists,Figure 4.11 shows the Grok operations to detect behaviour-alteration hotspots. This hotspot can be detected using the following steps:

1. First, create a set called `callbackFuncs` that contains functions that receive data from other features. This can be done by looking at the functions' `receiveFunc`



```
if (varA == true) {
    doSomethingInteresting();
} else {
    doSomethingBoring();
}
```

```
if (varB == true) {
    doSomethingInteresting();
} else {
    doSomethingBoring();
}
```

Figure 4.10: Two examples of the behaviour-alteration hotspot with two features: A and B.

```
1 callbackFuncs = receiveFunc . {true};
2 controlVars = isControlFlow . {true};
3
4 masterCalls = varAssign + call + varInfFunc + write;
5 masterCalls = masterCalls+;
6
7 behAlter = callbackFuncs o masterCalls o controlVars;
8 behAlter;
```

Figure 4.11: The Grok syntax for detecting the behaviour-alteration hotspot.

attribute. Line 1 of Figure 4.11 shows the Grok syntax to do this.

2. Next, create a set called `controlVars` that contains variables that are present in the decision condition of some control structure. This can be done by looking at the variables' `isControlVar` attribute. Line 2 of the Grok script shows this step.

3. Create a master relation called `masterCalls` that contains the union of the following relations: `call`, `write`, `varInfFunc`, and `varAssign`. This relation contains every single instance where a function or a variable transfers data flowing from domain to range. This step is shown in line 4 of the script.

4. Get the transitive closure of the `masterCalls` relation and store the results back in `masterCalls`. This relation now is a graph of the dataflow in the program. The transitive closure of `masterCalls` is obtained in line 5.

5. Apply the `callbackFuncs` set on the domain of the `masterCalls` relation using composition. Then, apply the `controlVars` set on the range of the `masterCalls`. Store these results back in `masterCalls`. Now, every entry in the `masterCalls` relation starts with a callback function and ends with a variable that affects a feature's control flow. Line 7 in the Grok script shows these join operations. Print the results.

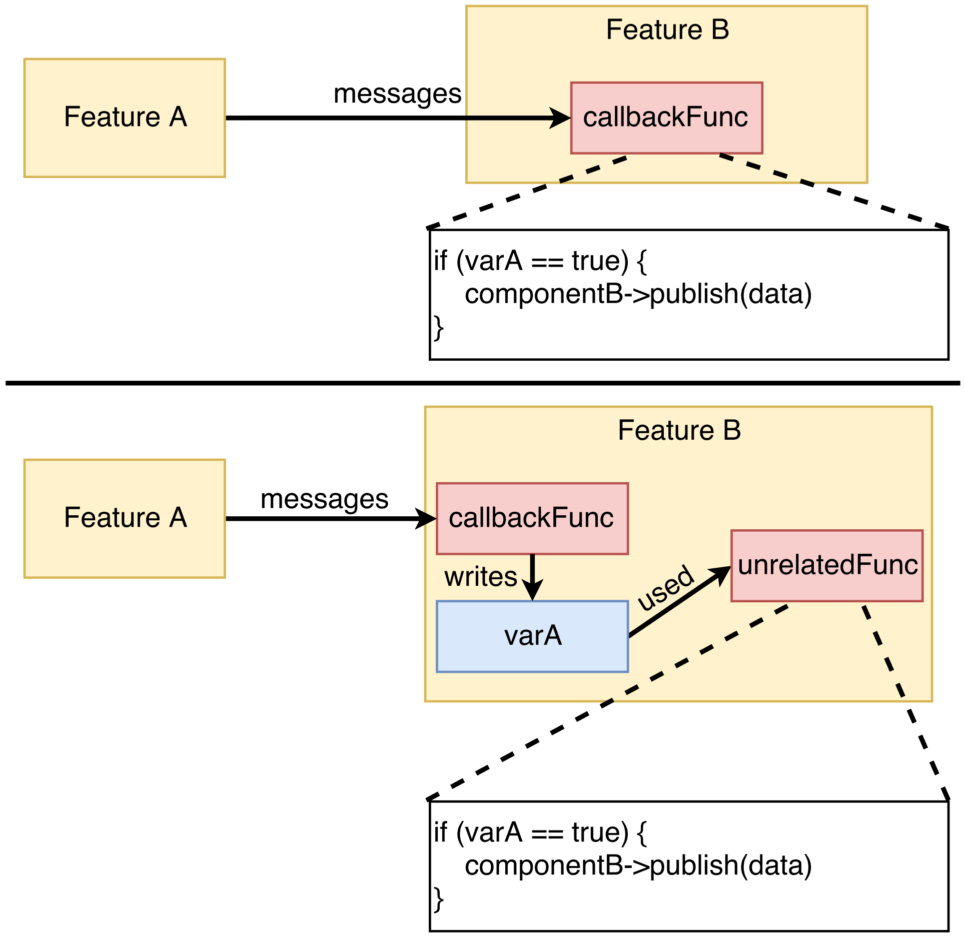
4.3.2 Publish-Alteration

The publish-alteration hotspot is a more specific version of the behaviour modification hotspot. This hotspot describes situations where a feature sends a message to another feature causing that feature to send a message to a third feature. This hotspot can occur in two ways. First, calls to "publish" messages to other features can be in a control structure that have variables in the decision portion that are modified directly or indirectly by a callback function or descendant function. This hotspot is similar to the behaviour-alteration hotspot. The other way this hotspot occurs is where a publish function call is directly inside the callback function or a descendant function called by that callback function. This means that if a message is received by that callback function, the publish call will automatically be activated. In both ways, if a feature receives a message to that callback function, this will likely result in the feature sending a message to another feature at some point in the future.

The benefit of detecting this hotspot is that it allows developers to see how a feature modifies the message-passing behaviour of another feature. Although it might be easy to determine which features message each other, determining whether features affect how another feature communicates. For instance, although ROS provides a visualization tool called `rqt_graph` that dynamically shows which features message other features, there is no way to see which features have influenced other features. Further, a feature that affects another feature's message-passing might introduce unexpected interactions between features.

Figure 4.12 shows two instances of this hotspot in a program with two features: A and B. In the first example, feature A messages feature B which causes the callback function `callbackFunc` to immediately publish data to a third feature. In the second example, feature A messages feature B which causes the callback function `callbackFunc` to write to a variable called `varA`. This variable is then used in a function called `unrelatedFunc` in a control structure that affects whether feature B publishes data to another feature.

Similar to the behaviour-alteration hotspot, detecting this hotspot requires a fairly detailed model. The bold elements in the subsequent list show additions to the model from the behaviour-alteration hotspot: (1) feature entities, (2) function entities, (3) variable entities, (4) a relation called `direct` that describes which features directly communicate with other features, (5) a relation called `call` that describes which functions call which other functions, (6) a relation called `write` that describes which functions write to which variables, (7) a relation called `varAssign` that describes which variables assign their values to other variables, (8) **a relation called `varInfluence` of the form `<VARIABLE>` `<PUBLISH_CALL>` that, for variables that participate in a decision condition of a control structure, highlights any publish calls that are nested within**, (9) a relation called `varInfFunc` of the form `<VARIABLE>` `<FUNCTION>` that for variables that participate in the decision condition of a control structure, highlights any function calls that are nested within, (10) an attribute called `isControlVar` which represents whether a variable is used in the decision condition of a control-flow structure, and (11) an attribute called `receiveFunc` which describes functions that receive messages from other features. Given such a model exists, Figure 4.13 shows the Grok operations to detect



```
if (varA == true) {
    componentB->publish(data)
}
```

```
if (varA == true) {
    componentB->publish(data)
}
```

Figure 4.12: Two examples of the publish-alteration hotspot for two features: A and B.

publish-alteration hotspots. This hotspot can be detected using the following steps:

1. First, create a set called `callbackFuncs` that contains functions that receive data from other features. This can be done by looking at the functions' `receiveFunc` attribute. Line 1 of Figure 4.13 shows the Grok syntax to do this.

2. Create a master relation called `masterCalls` that contains the union of the following relations: `call`, `write`, `varAssign`, `varInfluence`, and `varInfFunc`. This relation contains every single instance where a function or a variable transfers data or where a control structure is modified by a variable leading to another function call or publish message. Line 3 in the Grok script shows the syntax to achieve this.

3. Get the transitive closure of the `masterCalls` relation and store the results back in `masterCalls`. This relation now is a graph of the dataflow in the program. Line 4 shows the syntax to compute transitive closure.

4. Join the `callbackFuncs` set on the domain of the `masterCalls` relation. Join the domain of the `direct` relation on the range of the `masterCalls` relation. Store these results back in `pubAlter`. Every entry in the `pubAlter` relation starts with a callback function and ends with a feature that is messaged due to that callback function. Line 6 shows how to carry out these joins. Print the `pubAlter` relation.

```
1 callbackFuncs = receiveFunc o {true};
2
3 masterCalls = varAssign + varInfluence + varInfFunc + call + write;
4 masterCalls = masterCalls+;
5
6 pubAlter = callbackFuncs o masterCalls o direct;
7 pubAlter;
```

Figure 4.13: The Grok syntax for detecting the publish-alteration hotspot.

4.4 Chapter Summary

In this chapter, we introduced six types of hotspots that might suggest feature interactions in a message-passing automotive system. There were three different classes of feature interaction hotspots: *feature-communication*, *multiple-input*, and *control-flow* hotspots. The feature-communication hotspots discover areas in the software system where features communicate with each other. The multiple-input hotspots discover areas where multiple features publish messages that are received by the same feature resulting in a race condition. Lastly, the control-flow hotspots discover areas where the behaviour of a feature is changed as a result of a message received. For each hotspot, we present the steps required to detect it and the factbase elements required.

Chapter 5

Case Study

To determine the effectiveness of detecting feature-interaction hotspots using the relational algebra toolchain, a case study was conducted on the Autonomoose autonomous driving project. In this case study, Grok scripts were developed that detect each type of hotspot described in Chapter 4. Each of these scripts were then run on a TA model extracted from the Autonomoose source code and the results were analyzed.

Before the case study results are presented, a brief description of the Autonomoose autonomous driving project and its architecture and hardware components is provided in Section 5.1 provides a description of Autonomoose's architecture and hardware components. Section 5.2 presents the setup and results from the case study. Due to confidentiality obligations with Autonomoose developers, the description of Autonomoose and the case study are deliberately vague.

5.1 Autonomoose

Developed by the University of Waterloo Intelligent Systems Engineering (WISE) Lab, Autonomoose is a custom, autonomous software stack developed for a modified 2015 Lincoln MKZ. The goal of this project is to develop a vehicle that operates at level 4 of the SAE autonomous driving standard [51]. For a vehicle to be at this level, it needs to be in control of all aspects of the dynamic driving task and not require the human in the vehicle to respond to a request to intervene. Figure 5.1[1] shows the modified Lincoln MKZ used in the Autonomoose project.

[1]Image comes from [52].

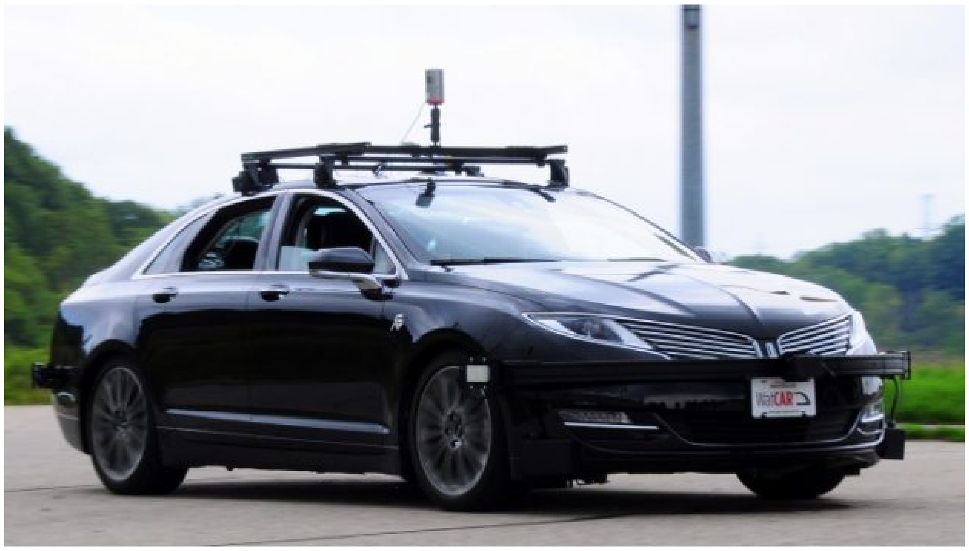


Figure 5.1: The modified Lincoln MKZ used by Autonomoose.

Similar to other autonomous driving projects, the Lincoln MKZ used by Autonomoose has a plethora of sensors that allow it to have an accurate model of its environment. The vehicle features a global positioning system (GPS) and inertial measurement unit (IMU) that together allow Autonomoose to discern the car's position on the road. The GPS has an accuracy of 5cm and updates at 20Hz and the IMU has a heading angle accuracy of 0.08 degrees and updates at 100Hz [53]. The vehicle is also equipped with a LIDAR system that allows it to detect hazards in all directions. Lastly, the vehicle has a built-in DSRC radio that provides it with the ability to communicate with other vehicles (V2V) and infrastructure (V2I).

The Autonomoose software stack uses the sensors to make decisions that are then applied to the vehicle's actuators. Figure 5.2 shows a high-level visualization of how the system architecture of Autonomoose is arranged. By obtaining data from sensors, the software generates a route plan that is executed by vehicle actuators. Importantly, the software is grouped into different features that each may be located on separate ECUs. Due to this, sensors, actuators, and features need to be able to communicate with each other in a reliable, fault-tolerant manner to ensure passenger safety. The Autonomoose software stack operates on top of the vehicle's existing software stack. Rather than using the vehicle's native CAN bus communication network, the Autonomoose software features communicate using the ROS framework as described in Section 2.3. By using a ROS-CAN bridge, information obtained from vehicle sensors can be used by Autonomoose and

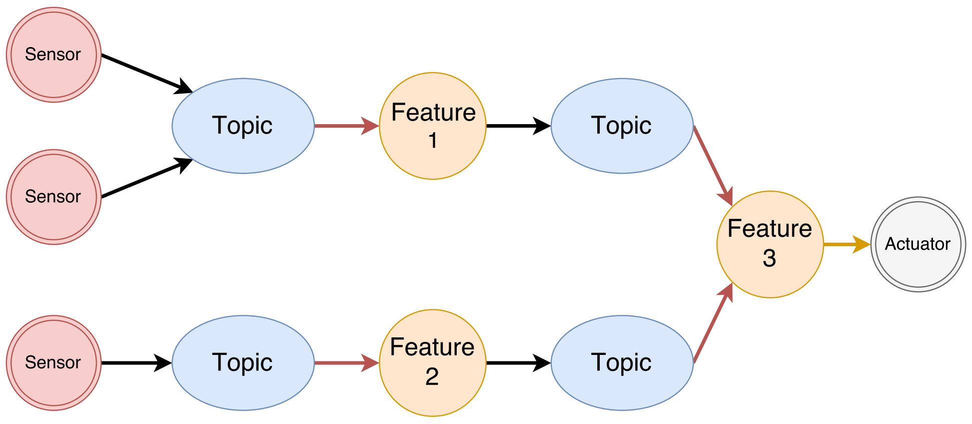


Figure 5.2: A high-level overview of the Autonomoose software stack.

commands from Autonomoose can be sent to vehicle actuators.

As mentioned before in Chapter 1, we define a feature as "a coherent and identifiable bundle of system functionality that helps characterize the system from a user perspective" [20]. Autonomoose features are grouped into separate ROS packages that each achieve a single goal. This configuration is beneficial for developers as it improves comprehension. Our case study benefits from this type of feature decomposition and organization: it eases model extraction in that features do not have to be explicitly identified in the source artifacts; each feature is identified with a specific ROS package. Because each ROS package is a separate artifact to be analyzed, the Rex fact extractor can automatically delineate each feature in the resulting model.

Autonomoose Statistics	
Lines of Code	74,215
Number of Features	14
Number of Functions	1,298
Number of Variables	5,321
Number of Communication Channels	14

Table 5.1: Code statistics for the Autonomoose project as of October 2017.

Table 5.1 shows several statistics regarding the Autonomoose codebase. Overall, there are over 74,000 lines of code in the entire project that we analyze including libraries bundled within ROS packages. These lines of code are distributed into 14 different features containing a total of 1298 functions and 5321 variables. Lastly, there are 14 unique communication channels that the features use to communicate. These communication channels are topics that have at least one publisher and subscriber connected to them.

5.2 Autonomoose Case Study

A case study was conducted on Autonomoose to determine whether the relational algebra toolchain could be used to detect feature-interaction hotspots. The case study evaluated each of the three feature-interaction hotspots were proposed in Chapter 4: (1) feature–communication hotspots, (2) multiple-influence hotspots, and (3) control-flow hotspots.

To detect each of these hotspots, the Autonomoose source codebase was analyzed using the Rex fact extractor to produce a queryable model. Hotspot definitions as described in Chapter 4 were used to generate Grok relational algebra scripts to detect these hotspots. Then, each hotspot script was run on the extracted model, and results were analyzed manually to check the TA model and results for completeness and correctness. As mentioned before, this thesis presents aggregated results from this case study; models and direct results have been simplified to comply with confidentiality agreements made with the Autonomoose developers.

The remainder of this section describes the case study in detail. Section 5.2.1 describes how the case study was setup and the models generated. Section 5.2.2 discusses the quality of the Autonomoose factbases extracted using the Rex extractor. Section 5.2.3 discusses feature-communication hotspots, Section 5.2.4 discusses multiple-influence hotspots, and Section 5.2.5 discusses control-flow hotspots. Lastly, Section 5.2.6 corroborates the results obtained with Autonomoose developers.

5.2.1 Setup

To detect hotspots in Autonomoose, some initial setup was required. First, we had to determine which entities and relations to include in the TA model so that all classes of hotspots could be detected. After examining the hotspot definitions from Chapter 4, Table 5.2 shows the entities and relations that need to be in an Autonomoose model to detect all hotspots. In this table, control-flow variables are variables that participate in the decision condition of a control-flow structure. The need to extract such information from Autonomoose was the motivation behind building the Rex extractor; information pertaining to ROS messages needed to be extracted as well as typical C++ information such as functions and variables.

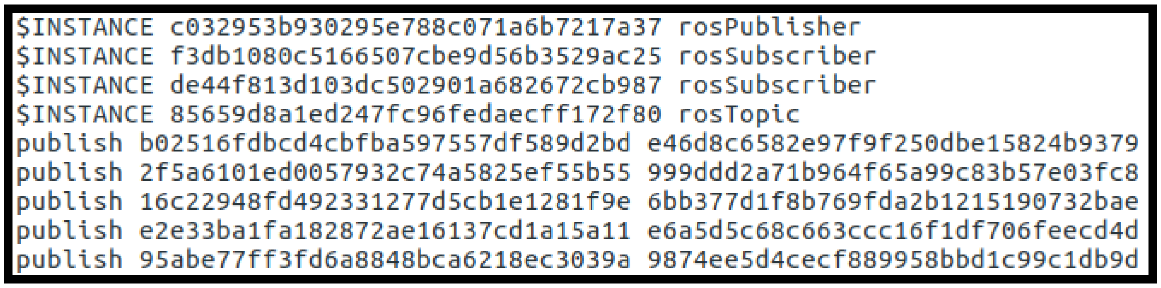
Entities	Relations	Attributes
• Features • Classes • Functions • Variables • Topics • Subscriber Nodes • Publisher Nodes	• A relation denoting data being sent from publisher to topic. • A relation denoting data being sent from topic to subscriber. • A relation denoting function calls. • A relation denoting a hierarchy of components. • A relation denoting variable writes. • A relation showing which control-flow variables result in publish messages. • A relation showing which control-flow variables result in function calls.	• A boolean attribute denoting whether a variable participates in the decision condition of a control-flow structure.

Table 5.2: Required model elements to detect all hotspots.

The Rex extractor was incorporated into Autonomoose's build toolchain so that Rex's Clang API could accurately parse the Autonomoose source code. Integrating Rex was challenging because ROS-based projects use Catkin as a build tool and use a number of different compiler flags that are hidden to the user. Catkin is difficult to add additional compiler flags that are required to use Rex. Furthermore, Rex relies on projects having a compilation database to receive compilation flags needed to build the project. Therefore, to use Rex on Autonomoose, Catkin needed to generate one.

To solve this problem, we modified one of the top-level build scripts for Autonomoose to contain a command that forces Catkin to generate a compilation database for each ROS project compiled; and Autonomoose had to be cleanly built without Rex to generate this database. By doing this, Catkin created a `compile_commands.json` file for each feature in Autonomoose. Because Rex needed to analyze all Autonomoose features at once, all the compilation databases were merged into a single file.

With a single compilation database created, Rex was able to analyze the entirety of



```
$INSTANCE c032953b930295e788c071a6b7217a37 rosPublisher
$INSTANCE f3db1080c5166507cbe9d56b3529ac25 rosSubscriber
$INSTANCE de44f813d103dc502901a682672cb987 rosSubscriber
$INSTANCE 85659d8a1ed247fc96fedaecff172f80 rosTopic
publish b02516fdbcd4cbfba597557df589d2bd e46d8c6582e97f9f250dbe15824b9379
publish 2f5a6101ed0057932c74a5825ef55b55 999ddd2a71b964f65a99c83b57e03fc8
publish 16c22948fd492331277d5cb1e1281f9e 6bb377d1f8b769fda2b121519073 2bae
publish e2e33ba1fa182872ae16137cd1a15a11 e6a5d5c68c663ccc16f1df706feecd4d
publish 95abe77ff3fd6a8848bca6218ec3039a 9874ee5d4cecf889958bbd1c99c1db9d
```

Figure 5.3: An example of the Autonomoose model generated by Rex.

Autonomoose's source code. To generate the models, all Autonomoose source files were added into Rex's processing queue and then all "test" files were removed. These test files are used for unit testing and are not important to the function of the software. Two different models were generated: `ANM_FULL.ta` and `ANM_MIN.ta`. The `ANM_MIN.ta` model is a lightweight model generated using Rex's simple-analysis mode. This version contains only ROS-based information about features, classes, and messages between classes. Although this model represents the entirety of the Autonomoose source code, it is only 60KB in size. The `ANM_FULL.ta` model is a detailed model generated using Rex's full-analysis mode. This model contains information about features, variables, functions, and ROS-based information such as topics and messages. This model is around 5MB in size. These models were generated using a snapshot of the Autonomoose project from October of 2017. In contrast, the total size of the source code is 900KB. Figure 5.3 shows a portion of the minimal model used in this case study. This figure shows several ROS components and several relationships between classes and ROS topics. Actual entity and relation names have been obfuscated using an MD5 hash.

With these models generated, Grok scripts for each hotspot were written to detect all instances of that hotspot in the source code. Appendix C shows each of the scripts used and provides a detailed description of all operations.

5.2.2 Rex Extractor Evaluation

As illustrated in Table 5.2, to create a model of the software for detecting feature-interaction hotspots, Rex extracts the following of different entities, relations, and attributes from the Autonomoose source code:

1. feature entities,

2. `class` entities,
3. `function` entities,
4. `variable` entities,
5. `publisher` entities,
6. `subscribers` entities,
7. `topics` entities,
8. `publish` messages,
9. `subscribe` messages,
10. function `calls`,
11. hierarchies between `classes` and `functions`,
12. functions that `write` to variables,
13. variables that assign their values to other variables called `varAssign`,
14. variables that influence whether a publish call is made called `varInfluence`,
15. variables that influence whether a function call is made called `varFuncInf`, and
16. an attribute that denotes whether a variable is used in the decision condition of a control statement called `isControlFlow`.

To assess the completeness and correctness of the extracted model, we compared the extracted entities, relations, and attributes in the model against the entities, relations, and attributes in the Autonomoose source code. To perform this verification, we manually examined the source code and for some datatypes identified a subset of instances of the above types of facts and for others identified all instances of the above types of facts to create a custom TA model. The entries in the Rex-generated model were compared to the manually-generated models to determine the model's precision and recall. Precision can defined as the number of all correct instances compared to the number of total reported instances and recall can be defined as the number of reported instances compared to the number of instances of that type. In other words, precision is the number of true positives

Element Type	Source Code	TA Model	Precision	Recall
Feature Entities*	14	14	100.00%	100.00%
`Class` Entities*	56	56	100.00%	100.00%
`Function` Entities	50	50	100.00%	100.00%
`Variable` Entities	166	166	100.00%	100.00%
Publisher Entities*	52	52	100.00%	100.00%
Subscriber Entities*	46	46	100.00%	100.00%
Topic Entities*	74	71	100.00%	95.90%
`Publish` Relation*	60	57	100.00%	95.00%
`Subscribe` Relation*	46	43	100.00%	93.48%
`Call` Relation	37	37	100.00%	100.00%
`Contain` Relation	76	76	100.00%	100.00%
`Write` Relation	287	226	100.00%	78.75%
`VarAssign` Relation	100	100	94.17%	97.00%
`VarInfluence` Relation*	52	52	96.23%	98.07%
`VarFuncInf` Relation	100	100	86.92%	93.00%
`isControlFlow` Attribute	164	164	88.51%	93.90%

Table 5.3: Completeness of entities, relations, and attributes in the Autonomoose model. The asterisk denotes relations where all entries of that type were analyzed.

over the sum of the true positives and false positives whereas recall is the number of true positives over the sum of the true positives and false negatives.

Table 5.3 shows precision and recall statistics for the Rex-based extraction of facts from Autonomoose source code. To evaluate the extraction of the non-ROS-based entities such as features, `classes`, and `functions`, we checked all feature and `class` entities and we checked 50 functions from four Autonomoose features and 166 variables from two navigation features. The 50 functions chosen for evaluation randomly. The 166 variables were obtained by selecting two random features and counting all variable instances inside. Rex has 100 percent precision and recall for all entities. For this evaluation, we checked all features and classes, 50 functions from four central Autonomoose features, and 166 variables from two navigation features were checked for precision and recall. As Rex gathers information directly from Autonomoose's AST , it would be extremely unlikely that our model would miss any of these entities or misrepresenting them in the output factbase.

To evaluate the extraction of ROS-based entities and relations, we checked all publisher, subscriber, and topic entities and relations across all 14 Autonomoose features. For publisher and subscriber objects, the Rex-generated TA model exactly reflected the code giving 100% precision and recall for Rex. For topic entities, Rex was able to extract this with 100% precision and 95.9% recall. Further, the `publish` and `subscribe` relations were not fully accurate as Rex had a recall of 95% and 93.48% respectively. This error occurs because Rex improperly extracts topic-name information from three topics in Autonomoose. The danger with inaccuracies in the extracted topic name is that Rex might not be able to properly link publisher and subscriber calls, causing communications between features to be omitted from the model. In our case study, there were three invalid topics in the Autonomoose model because in the source code the publishers and subscriber created these topics using a variable to hold the topic name, whereas Rex is only able to parse topic names that are string literals. This is a known limitation. Improper topic names affect only Rex's extraction of topic-based relationships but not the extraction of the `publisher`, `subscriber`, and topics themselves.

For the non-ROS-based relations for function `calls` and `contain` hierarchies, all 37 function calls were collected from two central Autonomoose features and then compared to the factbase and all 14 features were checked to ensure they "contain" all classes and two random Autonomoose feature classes were checked to ensure they "contain" all correct functions. It was found for these two relations that Rex has 100% precision and recall. As with the non-ROS-based entities, information about function `calls` or `contain` hierarchies can simply be pulled from the AST. Thus, the 100% precision and recall were expected.

To check the `write` relation for completeness, we checked all 44 callback functions in all 14 of Autonomoose's features and recorded whenever a variable assignment was made. We recorded which function each assignment was in and to which variable the assignment took place. These 44 functions represent just over 3% of Autonomoose's 1298 functions. Callback functions were chosen for analysis because these functions tend to write to many variables as they are responsible for transferring data received from topics. Compared to the 287 instances in the source code, in the Rex-generated TA model, there are 226 cases where a callback function writes to a variable. It was found that the precision was 100% due to the lack of false positives and the recall was 78.75%. The reason that the recall is imperfect is that the ability of Rex to detect variable reads and variable writes is limited. For one thing, statically detecting variable accesses in C++ with perfect precision and recall is impossible due to pointers and aliasing. For instance, if the address of one variable is written to a pointer inside a function, Rex would consider that to be a variable assignment despite the contents inside that pointer not being modified. A second limitation is that Rex does not detect cases where internal object fields are modified through function calls. For instance, if the `push_back` method in the `vector` class is used to add an object to that vector, Rex will not detect this instance. Despite the fact that the precision and recall of write detection can never both be 100%, future work should aim to improve this relation.

For the `VarAssign` relation, which denotes variables that assign their values to other variables, 100 variables out of Autonomoose's 5321 variables were randomly selected from a simple `grep` search. Then, the source code was manually analyzed to find reference to each of these variables and record whenever any of those variables were assigned to an expression that reference other variables. In other words, if the variable was on the left-hand side of an assignment statement, its use was recorded. The results of this analysis were compared to the Rex-generated TA model to determine the accuracy of `VarAssign`. Out of these 100 random variables that were tracked, the Rex was found to have a precision of 94% and recall and 97%. Three false negatives occurred because Rex ignores assignments in which variables are parameters in functions. For example, an expression like $var = function(varA, varB)$ is not reported even though $varA$ and $varB$ appear on the right-hand side of an assignment to var. As another example, this relation does not track the flow of data in a function call, from a variable used in a function call to its associated parameter in the called function. Future work should investigate developing a relation that can track this type of dataflow.

For the `VarInfluence` relation, which denotes which publish calls are located in control blocks that are altered by variables that affect that control flow statement's decision condition, there are only 52 publishers so all publisher objects and the associated `publish` calls were inspected. Each `publish` call was manually analyzed to determine whether it is made within the block of a control structure and which variables were directly referenced in the decision condition of that control structure. This was compared to the `VarInfluence` relation in the Rex-generated model. It was found that for 52 of the publishers, Rex has a precision of 96.23% and recall of 98.07%. This means that Rex accurately identified all cases where a specific variable modifies the decision condition of a control statement that a publish call is located in. There was one false negative case where the Autonomoose model was incorrect. This false negative was the result of a variable that is referenced in a function call that is used in an `if` condition expression. Thus, when a `publish` call is nested within an `if` statement whose condition is of the form if (`func`(`var`)), Rex ignores it even though the var technically affects this `if` statement.

For the `VarFuncInf` relation, which denotes which function calls are located in control blocks that are altered by variables that affect that control flow statement's decision condition, 100 of Autonomoose's 1298 functions were randomly selected for manual analysis. These 100 functions were chosen through the use of a random `grep` search. The analysis determined for each function whether there was a call to that function nested within a control structure and, if so, which variables are referenced in the decision condition of that control structure. The results of the manual analysis were compared to the `VarFuncInf` relation in the Autonomoose model. Rex had a precision of 86.92% and recall of 93%. For 100 of these variables, the model correctly identified 93 of them as being under a control structure or not. Of the 7 incorrectly identified functions, 2 were false positives (i.e., the model marks them as incorrectly called within a control structure) and 5 were false negatives. For the false positives, Rex correctly identified that the functions are called within control structures but incorrectly stated that some variable affects whether a `publish` call is made. Since this relation is extremely similar to the `VarInfluence` relation, false negatives occurred for the same reason. It resulted from a variable being referenced in a function call that is then used in an `if` condition expression.

For the `isControlFlow` attribute, which denotes which variables are part of the decision condition of control statements, all 167 variables inside ten randomly chosen classes were inspected. Each variable was manually inspected to determine whether it is referenced in the decision condition of a `for`, `switch`, `if`, or `while` statement. The results from this manual analysis were then compared to the Rex-generated model. Overall, for this attribute, Rex has a precision of 88.51% and a recall of 93.9%. All of the errors made were false positives: in certain complex statements, Rex cannot determine whether a variable is affecting the control flow or is simply part of a decision condition. For instance, in the statement if (`var` = `func(...)`), `var` is part of the statement, but it is not actually altering whether the `if` statement is taken or not.

5.2.3 Feature-Communication Hotspots

After evaluating the entities, relations, and attributes from the Rex extractor, the results for each hotspot type were obtained and then evaluated to determine the number of reported instances that describe likely feature interactions. For each of the three hotspot types, we present the number of reported instances of each hotspot type and evaluate the instances via manual analysis.

The first class of hotspots are *feature-communication* hotspots. The two hotspots in this class are *inter-component-based communication* and *loop detection*. The remainder of this section presents the results and verifies their correctness and completeness.

Results

The results of running the two Grok hotspot scripts are summarized in Table 5.4. For inter-component-based communication, there are 12 instances of direct communications and 3 instances of indirect communications between features. For instances involving direct communication, only seven features in Autonomoose actually send messages to other features. The reason there are few indirect instances despite Autonomoose's size is that Autonomoose is not designed to forward messages when a feature receives a message. Rather, each time an Autonomoose feature receives a message, it tends to change state which eventually causes it to publish a message to another feature. This is not an indirect inter-component-based communication instance because there needs to be a **continuous** chain of function calls from a feature receiving a message to sending a message. For situations where the feature changing state results in a publication, those are captured by hotspots in the control-flow hotspot class.

Hotspot Name	**Direct Results**	**Indirect Results**
Inter-Component-Based Communication	12	3
Loop Detection	1	1

Table 5.4: Result of detecting feature communication hotspots in Autonomoose.

For loop detection, there was one direct loop and one indirect loop detected. Both results are as expected given that the inter-component-based communication hotspot found a loop in the direct and indirect messages. The direct loop is interesting because this feature is a core part of Autonomoose and processes a lot of data. There are numerous reasons this loop could be present; it could be used to send data to other functions or be used as a heartbeat message.

Validation

As this class of hotspots contains all instances of all other hotspots, validating whether these hotspots detect potential feature interactions is not necessary. Instead, we validate the instances from all other hotspots and ensure that the results from this hotspot match the Autonomoose source code. For all 12 direct inter-component-based communication and single loop detection instances, we confirmed that these communications were present in the source code. This was expected because the accuracy of the Autonomoose factbase was already validated in Section 5.2.2. Other hotspot types are meant to detect interactions between features; their evaluation includes checking how accurate the reported hotspot instances are in predicting potential feature interactions.

5.2.4 Multiple-Input Hotspots

The second class of hotspots are *multiple-input* hotspots. Formally, there are two hotspots in this class: *multiple-publishers* and *race conditions*. The remainder of this section describes presents the results and verifies their correctness and completeness.

Results

After running the two Grok scripts to detect multiple-input hotspots, the following results were obtained which are summarized in Table 5.5. First, there are no multiple-publisher hotspots detected in the project. This means that any instance from a publishing feature has its own dedicated topic and recipient callback function. This result is as expected because one of the system's conventions is to have only one publisher per topic to reduce the risk of one feature overpowering another feature.

Hotspot Name	**Results**
Multiple Publishers	0
Race Condition	11

Table 5.5: Result of detecting multiple influence hotspots in Autonomoose.

Our race condition hotspot analysis detected eleven different variables for which possible race conditions were detected in assignments to these variables. Eight of those eleven variables are located in the same component. Furthermore, the number of callback functions that participate in the race condition depends on the variable. The majority of these variables have two callback functions writing to them and the variable with the most has four callback functions.

Validation

The multiple-publisher hotspot analysis detected zero instances of multiple topics triggering the same callback function; thus, the correctness of the reported instances cannot be validated. Since the extracted Autonomoose model was previous validated in this section, it is unlikely there are any false negatives as the `publish` and `subscribe` calls had a recall of 95% and 93% respectively. This means that only one `publish` and one `subscribe` call was missed.

Each of the eleven detected race condition hotspot instances was investigated manually for correctness. The result of this analysis is a categorization to the degree to which the

Hotspot	**Impossible**	**Unlikely**	**Probable**	**Total**
Race Condition	8	1	2	11

Table 5.6: The race condition hotspot in Autonomoose broken down by severity.

hotspot is likely to be a possible feature interaction worthy of further inspection. The three likelihood categories are: *impossible*, *unlikely*, and *probable*. The impossible category denotes instances that would not result in a feature interaction either from not fitting the hotspot definition due to errors in the Autonomoose TA model or because the updates to the variable in the race condition does not differ regardless of the feature that updates it. The unlikely category denotes instances that could potentially result in unexpected interactions between features but are unlikely to do so. Lastly, the probable category is used to denote instances that could result in potential interactions and should be inspected more thoroughly.

Table 5.6 shows the results of our classification. The *impossible* cases were instances where a variable in one feature received the exact same value from all callback functions that wrote to it. Although such instances fit the definition of multiple inputs leading to multiple assignments to the same variable, there is no undesired interaction because the fact that the callback functions all write the same value meaning that the effect of all the write operations is the same. The single unlikely instance is one where multiple callbacks update a time variable so that the recipient feature can record the time of the most recently received message. This time variable is used to determine when a timeout occurs. This instance is of little concern because, so long as *any* of these callback functions update the time variable, a timeout will not occur. Lastly, the two probable instances involve multiple callback functions that write to a variable that tracks the state of the car. This is important because unrelated functions use this variable to determine the behaviour of the car. As such, if conflicting messages are sent to these callback functions, there might be a potential for a state change to override other state changes.

5.2.5 Control-Flow Hotspots

The third class of hotspots are *control-flow* hotspots. There are two hotspots in this class: *behaviour-alteration* and *publish-alteration* hotspots. The remainder of this section presents the results and verifies their completeness and correctness.

Results

After running the two Grok scripts to detect control-flow hotspots, the following results were obtained which are summarized in Table 5.7. The instances column notes the number of detected instances for each hotspot. If a feature alters the behaviour of another feature multiple times, *each* will get recorded as a separate instance. The callback-functions column records the number of callback functions that are responsible for causing these instances. Note that a callback function may be responsible for numerous instances. Lastly, the direct and indirect columns note the number of features which directly and indirectly message the callback function of a target feature where a detected instance occurs. The reason the number of callback functions and direct communications do not match up is because some topics in Autonomoose have no features that publish to them[2].

Hotspot Name	**Instances**	**Callback Functions**	**Direct**	**Indirect**
Behaviour Alteration	1,368	35	11	16
Publish Alteration	64	22	7	15

Table 5.7: Result of detecting control flow hotspots in Autonomoose.

There are 1,368 instances of the behaviour-alteration hotspot where a feature's behaviour is altered due to messages received from a ROS topic. Although a large number of instances were reported, these instances occur in only 35 of Autonomoose's 47 callback functions. Furthermore, these callback functions are present in only 8 of Autonomoose's 14 features. Some of these instances likely represent intended interactions. For example four of features reported as having behaviour-alteration hotspots are essential in coordinating the vehicle's route. If one of these features receives a message, it is likely that feature will alter the vehicle's route based on a message received from another feature.

The publish-alteration hotspot analysis reported 64 cases where a feature publishes data because of a ROS message that it received. These 64 instances are spread across 22 callback functions. It was expected that this result would be lower than the behaviour alteration hotspot because there are not many `publish` calls in Autonomoose. Again, many of these instances occur in features that play a central role in the Autonomoose stack. These features receive data from a plethora of "vehicle-reporting" features, make decisions about the route based on this data, and pass those decisions on to other features.

[2]This primarily occurs in features at the top of the dataflow chain. These features receive messages directly from the vehicle through a ROS-CAN bus bridge.

Hotspot	**Impossible**	**Unlikely**	**Probable**	**Total**
Behaviour Alteration	1,021	225	122	1,368
Publish Alteration	12	27	25	64

Table 5.8: The behaviour-alteration and publish-alteration hotspots in Autonomoose broken down by severity.

Validation

Determining the correctness of these two hotspots is important. To do this, each reported instance of the two hotspots was manually validated and then classified into three categories: *impossible*, *unlikely*, and *probable*. As before, the impossible category denotes instances that would not result in a feature interaction either from not fitting the hotspot definition due to errors in the Autonomoose TA model or because it objectively does not result in a feature interaction. This could be because the behaviour change only affects local variables and a single function. The unlikely category is used to describe instances that fit the hotspot definition but are unlikely to result in unexpected interactions or impact the overall behaviour of the feature. Lastly, the probable category represents instances where the overall behaviour of the feature is impacted and where multiple features are involved. These instances should be investigated further.

Table 5.8 shows the results of classifying each behaviour-alteration instance. Of the 1,368 instances, 1,021 are impossible interactions, 225 are unlikely interactions, and 122 are probable interactions. Although the number of impossible instances is high, 17% of these are due to Autonomoose's use of ROS logging and debugging macros. After preprocessing, these macros expand into several lines of code that make use of variables that participate in the decision conditions of control structures. As such, if any callback function uses such a macro, this hotspot analysis will report the macro as an instance of behaviour alteration. Given that these macros are simply responsible for logging data to console and altering the logging frequency, it can be concluded that they do not result in feature interactions. The remainder of the impossible instances are false positives that either resulted from errors in the TA model or involve local variables that only affect minor control-flow statements. For the instances that arose due to errors, those occured from Rex incorrectly detecting a variable assignment or from Rex incorrectly stating that a variable was used in the decision condition of a control-flow statement. Regarding the minor local variables, there were situations where a local variable was assigned a value in a callback function and then, in that callback function, the local variable affected how an operation was performed. Regardless of what value that variable was assigned, the operation would be performed and the feature would not change state. Future versions of Rex should attempt to detect macro expansions and local variable modifications and ignore them to avoid overwhelming the user.

For a behaviour-alteration hotspot instance to be classified as unlikely, the alteration needs to affect a feature's behaviour only slightly. For instance, a large number of these cases are variables that participate in the terminating condition of a `for` loop. In other cases included in this classification, one variable affects what the value of another variable is. In all of these cases, there is no major behaviour change in the node or function. As such, many of these can probably be ignored.

For a behaviour-alteration hotspot instance to be considered probable, it has to include an assignment to a variable that goes on to modify the state of the feature. In many of these cases, the variables modified are fields inside the feature that maintain its state and alter which operations the feature performs. It is recommended that each of these 122 instances should be investigated further.

Table 5.8 shows the results of classifying each publish-alteration instance. Of the 64 instances reported, 12 are impossible, 27 are unlikely, and 25 are probable. Of the 12 impossible instances, the majority are merely used for debugging, to allow developers to see real-time information about the vehicle. The publish call involved in each of these instances publishes to a debugging topic. The other instances classified as impossible are false positives that result from errors in the Autonomoose model. As in the behaviour-alteration results, most of these false positives occur from errors in how Rex determines variable writes and control-flow variables.

The 27 instances that are deemed to be unlikely interactions are instances in which the callback functions do not have any features that message them. Because they subscribe to named ROS topics, these instances get reported as a publish-alteration instance. However, these callback functions receive information from the car's sensors rather than from other features hence they are unlikely to indicate undesired interactions. It is possible these could still be problematic since sensor data sent from the vehicle sensors could still possibly negatively react with these features.

Lastly, the 25 probable, publish-alteration hotspot instances have publish calls involved to pass "important" messages to recipient features. Important messages include route information or information about vehicle state. Investigating these reported instances further using manual analysis is important as they are critical to how Autonomoose operates.

5.2.6 Valdiation with Autonomoose Developers

The fact that we evaluated the utility of reported hotspots by manually analyzing the identified code and classifying identified hotspot instances as being impossible, unlikely, or probable introduces an external threat to validity. To mitigate this threat, we verified a subset of these instances and classifications with developers from the Autonomoose project. Specifically, a subset of the reported instances that were classified as unlikely or probable interactions were presented to a domain expert, Dr. Michal Antkiewicz, who is the lead research engineer on the Autonomoose project, for his assessment[3]. Because the race-condition hotspot had only three non-impossible instances, all three were presented. For each instance, our domain expert determined whether he thought it was valid, and if so, classified it as either *unlikely* or *probable*.

Race Condition

All three non-impossible instances of the race condition hotspot were shown to the domain expert. He characterized two as probable and one as unlikely. This classifications matched the ones that we made as reported in Section 5.2.4.

For the two probable instances, the domain expert confirmed that there was some global variable that was modified by multiple callback functions that went on to affect the feature's state. He noted that for these cases, it would be useful to have an engineer examine those callback functions more thoroughly.

Behaviour Alteration

We randomly selected 10 out of 145 non-impossible instances of reported behaviour-alteration hotspots and showed them to our domain expert. We had previously manually classified these ten instances and labeled six as being unlikely and four as being probable interactions. Interestingly, the domain expert found that all the instances presented were not really useful. Based on the instances presented to him, he determined that this hotspot was not very useful.

One reason that might explain why this type of hotspot was found not to be useful is that our expert was shown only ten random instances out of 145. Perhaps there were

[3]Impossible instances are cases that do not fit the hotspot definition as they are errors that result from impossible entries in the Autonomoose TA model. As such, these are not useful to present to our domain expert.

some useful instances among the 145 but they were not among the instances shown to the Autonomoose developer. If so, this is a concern because having this much noise in the results might mean that developers could miss an important hotspot instance. Future work should investigate how to prioritize the hotspot analysis results to attempt to focus developers' attention on the hotspot instances that are most likely to be associated with undesired interaction. Furthermore, many of the probable instances had a simple path from callback function to control variable. It would be interesting to present instances involving more complex paths to our domain expert.

Publish Alteration

We randomly selected 10 out of 53 non-impossible instances of reported publish-alteration hotspots and showed them to our domain expert. We had previously manually classified five of these as unlikely interactions and five as probable interactions. In contrast, our expert categorized six as probable and four as unlikely. Of his classifications, his matched with ours for three probable cases and two unlikely cases.

The reason that the Autonomoose developer and our classifications differed is that our expert was interested only in those publish-alteration instances where there was a complex path between the origin callback function and the destination publish call. For instance, if a callback function directly publishes a message, this was considered to be a "known", expected, and desired interaction and not considered interesting. However, if a callback function calls another function, which writes to a variable, which then affects whether a publish call is made, this would be considered a possible unintended flow of information leading to an undesired interaction worth of further inspection. Many of these instances with complex traces would be difficult for a developer to detect without the use of tools.

5.3 Threats To Validity

A major threat to the internal validity of this case study is subjectivity caused by us manually classifying reported hotspot instances. By performing this classification ourselves, we could have introduced our own bias because we have a stake in the results. Furthermore, although we have knowledge of the Autonomoose software architecture, we are not system domain experts and we could not classify the results without bias. The purpose of presenting a subset of these reported instances to the Autonomoose engineers was to reduce this bias. However, only ten hotspot instances of each type were presented. This means that not all classifications were verified by domain experts.

The primary threat to external validity in this case study is that the results are not necessarily generalizable to other software systems or even to other automotive systems. As discussed previously, Autonomoose is not a typical automotive system as it uses the ROS architecture to facilitate communication between features as oppose to the CAN architecture.

5.4 Chapter Summary

In this chapter, we presented a case study on the Autonomoose autonomous car project. We generated a tuple-attribute factbase that describes the Autonomoose codebase, tested the precision and recall of the factbase, and ran hotspot Grok scripts on the factbase to detect potential feature-interaction hotspots. Overall, we found that, for most entities, relations, and attributes in the factbase, the precision and recall of the model are over 90 percent. The results obtained in this case study were evaluated by Dr. Mikael Antkiewicz, Autonomoose research engineer. While credible hotspot instances were discovered, the recall of Rex model elements should be improved in future work to ensure to avoid false negatives.

Chapter 6

Conclusions

This chapter discusses the contributions made in this thesis as well as the limitations of our work and areas of future work that should be pursued.

6.1 Contributions

Our work presents four major contributions:

- The development of the *ClangEx* fact extractor which can extract static design information about AST-based entities and their dependencies from C and C++ source code. ClangEx was written as a general purpose extractor, to be used as a starting point to develop more specific C and C++ fact extractors.
- The development of the *Rex* fact extractor, which is used to extract information about ROS messages sent between components. This extractor allows for distributed, message-passing systems to be analyzed using the relational algebra toolchain. This extractor was built from ClangEx.
- The identification of six feature-interaction hotspots that might be present in distributed, message-passing systems. Each of these hotspots can be detected in models generated by the Rex extractor. These six hotspots are: *inter-component-based communication*, *loop detection*, *multiple publishers*, *race conditions*, *behaviour alteration*, and *publish alteration*.

- A case study that evaluates the feasibility and utility of detecting these hotspots in an automotive software system. The case study was conducted on the Autonomoose autonomous car software project which contains over 20,000 lines of code in the main project [54]. Each hotspot type was run on the Autonomoose TA model and the instances were classified into levels of likelihood of being a feature interaction of interest. Furthermore, Dr. Michal Antkiewicz of the Autonomoose project classified a subset of these reported instances using the same classification scheme.

6.2 Limitations

Although the our toolchain is effective at detecting feature-interaction hotspots in distributed, message-passing systems, there are several limitations to this methodology. The majority of these limitations are due to limitations of static analysis which impact the precision and recall of the fact extractors.

Although fact extractors are capable of parsing and modeling the entire codebase of a project, these extractors share limitations that affect other static-analysis tools. Static analysis examines code without executing it, which means that some code behaviours such as threads, message-passing, or order of execution of statements cannot always be modelled accurately. This leads tools to reason about the system in an approximate manner [55]. Interactions shown in models developed using fact extractors might not actually be in the running software. Due to these drawbacks, detecting feature interactions using this approach is not a fully automated process: there is a strong possibility of false positives and negatives so the main contribution of this toolchain is to reduce the amount of manual inspection that needs to be done on the software system.

Another limitation is that the Rex extractor relies on several "programmer-conventions" when generating a model of message-passing information in a ROS project. Relying on these conventions was sufficient to analyze Autonomoose, but Rex may not be as successful in generating a correct model for other ROS-based projects. For example, Rex assumes that the topic name that is passed to a publisher or subscriber object is a string literal, whereas it is legal in ROS to pass topic names as string variables to a publisher or subscriber[1].

Lastly, there are limitations on the extractors' ability to detect variable reads and writes in C and C++. Due to aliasing, developing a precise variable read and write detector is

[1]Determining the contents of a variable statically is extremely difficult if not impossible. Future work on Rex might be able to determine the variable contents and use those contents for the topic name in *simple* cases.

impossible in static analysis because variable values can be used as addresses or values. The read and write detection in Rex and ClangEx is simplistic: all binary and unary expressions involving variables are treated as being non-aliased. This means that even if the indirection or address operator is used, Rex will ignore those operators and still record the reference. Of course, if a project had strict coding standards that dictated how aliased variables or topic names were used, it is expected that the precision and recall of the Rex-generated TA models would increase.

6.3 Future Work

Future work falls into four main categories: (1) determining the feasibility of using the relational algebra toolchain on automotive source code that utilizes the CAN bus protocol, and conducting a case study on this type of software; (2) updating ClangEx and Rex to extract more information from C and C++ source code and to improve current detection; (3) exploring whether analyzing additional software artifacts can improve hotspot detection quality; and (4) developing more hotspots and creating a method to automatically classify the likelihoods that hotspot instances represent feature interactions of interest. Each of these areas of future areas of work are explained further below.

First, this thesis examines a distributed automotive system where features are separated into different modules and communicate using a common framework, but this system is not fully indicative of other automotive systems. The majority of automotive systems tend to be complexly distributed and use a centralized CAN bus to communicate. A lot of work is still required to determine how feature-interaction information can be extracted from CAN bus systems. Furthermore when analyzing software built on top of the AUTOSAR framework, the extractors targeting traditional automotive systems must be able to understand the AUTOSAR API. Once this has been achieved, it is important to conduct a case study using an AUTOSAR and CAN-based automotive system to evaluate whether the hotspots postulated in this thesis are applicable. Although AUTOSAR is not currently used in many traditional automotive systems, most automotive companies have indicated that they intend to to move to AUTOSAR in the future [56]. An advantage of focusing on AUTOSAR in future work is that this framework divides the software system into multiple layers where each layer provides strong abstractions to the layer above. This layered architecture would make analyzing an AUTOSAR-based project far easier than analyzing another automotive software projects due to a standard, unified API. Analyzing messages passed between ECUs or services is likely to be simpler than analyzing raw CAN bus messages.

Another area of future work is to further develop the ClangEx and Rex extractors to improve their accuracy and to add more TA model elements. Although the Rex extractor is capable of extracting enough information to detect all the hotspot types proposed in this thesis, extracting further entities and relations from the target source code could improve hotspot detection and could allow for the detection of more complex hotspots. As an example, one new relation that could be developed would track how data moves between function calls. For instance, if there is a function call such as $var = function(varA, varB)$, it would be important to record which parameters $varA$ and $varB$ transfer their data to and to record that $function$ returns data to var. For ROS-based projects, entities could be added to record additional ROS components such as timers[2]. These could allow users to track which variables are influenced by timers or services. Furthermore, although most of the relations in Rex were verified to have over 90% accuracy, it would be good to further improve their accuracy to produce better models.

As a third area of future work, it would be interesting to develop extractors and tools that can extract facts from other types of *structured* artifacts as well as from C and C++ code. By generating models that include information about code and other project files, queries could be written that are more precise and less erroneous. For instance, Autonomoose makes use of XML scripts that configure the software stack. These configuration scripts dictate which nodes (features) are started and how many of each node type to start. Without processing these files, models of Autonomoose contain a "full-project" view of the codebase where all features are active at the same time. This could result in detected interactions that might **never** be present in an actual run of Autonomoose. In addition, by including information extracted from configuration scripts, queries could find instances where the same feature running multiple times may write to the same variable.

Last, although the collection of hotspots presented in this thesis is thorough, these hotspots are not exhaustive. Future work should investigate developing more hotspots to better detect feature interactions in message-passing systems.For example, some platform-specific hotspots could be introduced. ROS has the concept of timers, which call specific functions at a particular frequency; developing hotspots that involve those might be useful (e.g., users could discover when a feature publishes data as a result of a timer). As another example, it might be beneficial to develop a method to filter or prioritize hotspot results to focus the users attention on the most promising reported hotspots. For instance, the behaviour-alteration hotspot analysis reported over 650 instances in the Autonomoose software and many of these reported instances were deemed not useful by Autonomoose research engineers. Being inundated by so many results may lead engineers to be over-

[2]As per ROS documentation, timers call a certain function with a certain frequency. This could be used to send data to other features at a set frequency.

whelmed and overlook some important instances. Prioritizing results might involve identifying hotspots based on predefined patterns of false negatives, or might consider the trace length of an instance when evaluating how likely the hotspot is to be a feature interaction of interest. For instance, for the publish-alteration hotspot, an instance where a callback function immediately publishes data is more likely to be an intended interaction and to be less interesting than an instance where a callback function writes to a variable that then affects whether data is published.

References

[1] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh, “An efficient, robust, and scalable approach for analyzing interacting android apps,” in *Proceedings of the 39th International Conference on Software Engineering*, pp. 324–334, IEEE Press, 2017.

[2] R. Purandare, J. Darsie, S. Elbaum, and M. B. Dwyer, “Extracting conditional component dependence for distributed robotic systems,” in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 1533–1540, IEEE, 2012.

[3] A. Bridgwater, “Winding up klocwork source code analysis,” oct 2013.

[4] R. N. Charette, “This Car Runs on Code,” *IEEE Spectrum*, Feb. 2009.

[5] E. Priestley, “How many lines of code is facebook?,” jan 2011.

[6] M. Windows, “Windows - posts,” jan 2011.

[7] G. Clarke, “Cern’s boson hunters tackle big data bug infestation,” sep 2011.

[8] S. Edelstein, “The ford gt has more lines of code than a boeing passenger jet,” may 2014.

[9] OpenHub, “The android open-source project,” 2017.

[10] R. Paul, “Linux kernel in 2011: 15 million total lines of code and microsoft is a top contributor,” apr 2012.

[11] F. M. JR., “Excitement and dismay at space telescope center,” feb 1989.

[12] C. Metz, “Google is 2 billion lines of codeand its all in one place,” sep 2015.

[13] W. Platz, “Software fail watch: 2016 in review,” whitepaper, Tricentis, 2017.

[14] W. Bank, “Mexico gdp per year,” 2017.

[15] S. Borland, “Up to 300,000 heart patients may have been given wrong drugs or advice due to major nhs it blunder,” may 2016.

[16] N. G. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” *Computer*, vol. 26, pp. 18–41, July 1993.

[17] T. Hummel, M. Kühn, J. Bende, and A. Lang, “Advanced driver assistance systems,” *German Insurance Association Insurers Accident Research. Available on www. udv. de, accessed at*, vol. 6, no. 01, p. 2015, 2011.

[18] N. Bomey, “Fiat chrysler recalling 1.25m ram pickups to fix rollover air bag, seat belt failure,” may 2017.

[19] A. L. Juarez Dominguez, *Detection of feature interactions in automotive active safety features.* PhD thesis, University of Waterloo, 2012.

[20] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, “A conceptual basis for feature engineering,” *Journal of Systems and Software*, vol. 49, no. 1, pp. 3 – 15, 1999.

[21] S. Arora, P. Sampath, and S. Ramesh, “Resolving uncertainty in automotive feature interactions,” in *2012 20th IEEE International Requirements Engineering Conference (RE)*, pp. 21–30, Sept 2012.

[22] J. Cohen, “11 proven practices for more effective, efficient peer code review,” jan 2011.

[23] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world,” *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[24] V. Okun, A. Delaitre, and B. P. E., “Report on the static analysis tool exposition (sate) iv,” in *NIST Special Publication 500-297*, 2013.

[25] C. W. CLEVERDON, “On the inverse relationship of recall and precision,” *Journal of Documentation*, vol. 28, no. 3, pp. 195–201, 1972.

[26] C. Willis, “Cas static analysis tool study overview,” in *proc. eleventh annual high confidence software and systems conference*, p. 86, National Security Agency, 2011.

[27] R. C. Holt, “Introduction to the grok programming language,” *University of Waterloo*, 2002.

[28] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems.* Springer Science & Business Media, 2006.

[29] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, “Detecting event anomalies in event-based systems,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 25–37, ACM, 2015.

[30] D. Kozen, “Kleene algebra with tests and the static analysis of programs,” tech. rep., Cornell University, 2003.

[31] SciTools, “Scitools’ understand,” 2017.

[32] SciTools, “Writing codecheck scripts,” 2017.

[33] C. Bolduc, “Lessons learned: Using a static analysis tool within a continuous integration system,” in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 37–40, Oct 2016.

[34] S. A. G. (SWAG), “Javex - java fact extractor,” apr 2010.

[35] S. A. G. (SWAG), “Cppx - open source c++ fact extractor,” jun 2001.

[36] S. A. G. (SWAG), “Ldx and bfx pipeline,” 2010.

[37] S. A. G. (SWAG), “Asx - c/c++/assembler fact extractor,” jan 2017.

[38] J. Wu, *Open Source Software Evolution and Its Dynamics.* PhD thesis, University of Waterloo, 2006.

[39] R. Holt, “The tuple-attribute (ta) language,” 1997.

[40] A. Tarski, “On the calculus of relations,” *The Journal of Symbolic Logic*, vol. 6, no. 03, pp. 73–89, 1941.

[41] J. Uhl, “Rigi standard format,” 1996.

[42] R. C. Holt, “Structural manipulations of software architecture using tarski relational algebra,” in *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, pp. 210–219, Oct 1998.

[43] D. Beyer, A. Noack, and C. Lewerentz, "Simple and efficient relational querying of software structures," in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pp. 216–225, IEEE, 2003.

[44] N. Synytskyy, R. C. Holt, and I. Davis, "Browsing software architectures with lsedit," in *13th International Workshop on Program Comprehension (IWPC'05)*, pp. 176–178, May 2005.

[45] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, 2009.

[46] S. Agarwal, "How to use irobot create with ros indigo and gazebo," feb 2015.

[47] H. Fahmy and R. C. Holt, "Software architecture transformations," in *Proceedings 2000 International Conference on Software Maintenance*, pp. 88–96, 2000.

[48] I. J. Davis, M. W. Godfrey, R. C. Holt, S. Mankovskii, and N. Minchenko, "Analyzing assembler to eliminate dead functions: An industrial experience," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 467–470, March 2012.

[49] M. Jones, "Gcc hacks in the linux kernel," nov 2008.

[50] J. D. Herbsleb, "Global software engineering: The future of socio-technical coordination," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 188–198, IEEE Computer Society, 2007.

[51] S. O.-R. A. V. S. Committee *et al.*, "Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems," *SAE Standard J3016*, pp. 01–16, 2014.

[52] L. Mathews, "Autonomoose is the first driverless car on canadas roads," nov 2016.

[53] A. Dakibay, "Autonomous driving: Baseline autonomy," Master's thesis, University of Waterloo, 2017.

[54] J. Kuehn, "Ros code quality," Mar 2013.

[55] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24 – 27, 2003.

[56] F. Kirschke-Biller *et al.*, “Autosar–a worldwide standard current developments, roll-out and outlook,” in *15th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany*, 2011.

APPENDICES

Appendix A

Installing & Using ClangEx/Rex

A.1 Installing ClangEx & Rex

ClangEx and Rex are two separate C and C++ fact extractors that utilize similar libraries. Due to this, this section will provide information that installs the required libraries and tools used to install both extractors. Before they can be built, ClangEx and Rex both require the following to be installed on the target system: **CMake** 3.0.0 or greater, **Boost** 1.6 or greater, **LibSSL**, and **Clang** 5.0 or greater. CMake is used to build ClangEx and Rex, Boost is a collection of C++ libraries that, in the case of ClangEx and Rex is used to process command line arguments and directory information, and Clang is used to obtain AST information about the source code currently being processed. The Clang API provides methods for operating on C/C++ language features and carries out the brunt of the C and C++ analysis.

The remainder of this section provides information on how to install these required libraries and then how to build ClangEx and Rex from source. These instructions are for Linux-based systems but can be adapted to work on Windows or Mac OSX. Section A.1.1 describes how to install CMake, Boost, and Clang. If any of these are already installed on the target system, it can be skipped. Section A.1.2 describes how to build ClangEx from source and Section A.1.3 describes how to build Rex from source.

A.1.1 Prerequisites

CMake

If using a Ubuntu or Debian-based system, installing CMake is as simple as using `apt-get`. This is done using the following two commands:

```
$ sudo apt-get install cmake
$ cmake --version
```

Using `apt-get` might not install the latest version of CMake. However, as long as it installs a version of CMake greater than `3.0.0`, it can be used to build ClangEx and Rex. If this is the case, proceed to the Boost installation instructions.

If this method did not work, CMake must be built from source. To do this, the latest version of CMake source needs to be downloaded from the CMake website, compiled, and then installed. This guide provides instructions on how to build CMake `3.7.0` from source. The first step is to download the CMake source code and unzip it. This can be from the command line using the following commands:

```
$ wget https://cmake.org/files/v3.7/cmake-3.7.0.tar.gz
$ tar xvzf cmake-3.7.0.tar.gz
$ cd cmake-3.7.0
```

Once in the `cmake-3.7.0` directory, CMake can be configured and install on the target system. This process may take several minutes. To do this, use the following commands:

```
$ ./configure
$ make
$ make install
$ cmake --version
```

If these steps completed successfully, CMake is now installed and ready for use.

Boost and LibSSL

Installing Boost on Ubuntu or Debian-based systems is extremely easy by using the `apt-get` package manager. This installation process will install the libraries and add them to the target system's default *include* path. This can be done using the following command:

```
$ sudo apt install libboost-all-dev
```

Be sure that Boost version **1.6** or greater is installed or else ClangEx/Rex will not compile.

In addition, LibSSL is required. This can also be done using the `apt-get` package manager. This can be done using the following command:

```
$ sudo apt install libssl-dev
```

Clang

Although Clang exists as a package that can be installed using `apt-get`, it needs to be installed from source to take advantage of Clang's API. To do this, source code needs to be checked out from the official Clang repository, compiled, and then installed. The first step is to checkout the Clang source code using Subversion. This can be done by executing the following:

```
$ svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
$ cd clang/tools
$ svn co http://llvm.org/svn/llvm-project/
                    clang-tools-extra/trunk extra
$ cd ../../../..
```

These commands will download the LLVM and Clang source code to a directory called `llvm` in the current working directory. Now, Clang can now be built. This process can take up to several hours and uses a large amount of disk space. This guide shows how to build Clang in a directory called `Clang-Build` that is adjacent to the `llvm` directory. To use another directory, simply replace the `Clang-Build` string in the following commands with another directory name.

The following commands build Clang in the `Clang-Build` directory:

```
$ mkdir Clang-Build
$ cd Clang-Build
$ cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
        -DLLVM_ENABLE_EH=ON -DLLVM_ENABLE_RTTI=ON ../llvm
$ make
$ make install
```

Once these steps have been completed, Clang and LLVM have been installed. By typing `clang --version`, Clang should report that it is version `5.0` or higher.

A.1.2 Building ClangEx

To build ClangEx, the source code needs to be checked out from the ClangEx GitHub repository and then built using CMake. If ClangEx fails to build, first ensure that all tools and libraries installed in the previous section are present on the target system.

To download ClangEx, you first need to checkout the source code from GitHub. Before checking out the repository, ensure that you are in a directory where you want to install ClangEx. To checkout the ClangEx code from GitHub, execute the following command:

```
$ git checkout https://github.com/bmuscede/ClangEx.git
```

If this completed successfully, a directory called `ClangEx` will be created that contains all ClangEx source code.

Next, before ClangEx can be built, two separate environment variables must be set: `LLVM_PATH` and `CLANG_VER`. The `LLVM_PATH` variable tells ClangEx where LLVM and Clang were built to. The `CLANG_VER` variable is the version of Clang installed. To set these variables, open up `.bashrc` located in the home directory and add the following lines to the bottom of the file:

```
$ export LLVM_PATH=<PATH_TO_CLANG-BUILD>
$ export CLANG_VER=<VERSION_OF_CLANG>
```

Restart the terminal to ensure these variables have been exported.

The next step is to build ClangEx. These steps install ClangEx in a directory called `ClangEx-Build` adjacent to the `ClangEx` directory. To install ClangEx somewhere else, simply replace the `ClangEx-Build` string in the following commands with a desired directory name. The following steps install ClangEx in the `ClangEx-Build` directory:

```
$ mkdir ClangEx-Build
$ cd ClangEx-Build
$ cmake -G "Unix Makefiles" ../ClangEx
$ make
```

By running these commands, CMake will build the source code and an executable called *ClangEx* will be created inside the `ClangEx-Build` directory. To verify that ClangEx built correctly, run:

```
$ ./ClangEx
```

If ClangEx built, this command will run the ClangEx executable. This will print a splash screen and information to the screen. Now, ClangEx is ready to process C/C++ source files and generate tuple-attribute models.

A.1.3 Building Rex

To build Rex, the source code needs to be checked out from the Rex GitHub repository and then built using CMake. If ClangEx fails to build, first ensure that all tools and libraries installed in the previous section are present on the target system.

To download Rex, you first need to checkout the source code from GitHub. Before checking out the repository, ensure that you are in a directory where you want to install Rex. To checkout the Rex code from GitHub, execute the following command:

```
$ git checkout https://github.com/bmuscede/Rex.git
```

If this completed successfully, a directory called `Rex` will be created that contains all Rex source code.

Next, before the Rex extractor can be built, two separate environment variables must be set: `LLVM_PATH` and `CLANG_VER`. The `LLVM_PATH` variable tells Rex where LLVM and Clang were built to. The `CLANG_VER` variable is the version of Clang installed. To set these variables, open up `.bashrc` located in the home directory and add the following lines to the bottom of the file:

```
$ export LLVM_PATH=<PATH_TO_CLANG-BUILD>
$ export CLANG_VER=<VERSION_OF_CLANG>
```

Restart the terminal to ensure these variables have been exported.

The next step is to build Rex. These steps install Rex in a directory called `Rex-Build` adjacent to the `Rex` directory. To install Rex somewhere else, simply replace the `Rex-Build` string in the following commands with a desired directory name. The following steps install Rex in the `Rex-Build` directory:

```
$ mkdir Rex-Build
$ cd Rex-Build
$ cmake -G "Unix Makefiles" ../Rex
$ make
```

By running these commands, CMake will build the source code and an executable called *Rex* will be created inside the `Rex-Build` directory. To verify that Rex built correctly, run:

```
$ ./Rex
```

If Rex built, this command will run the Rex executable. This will print a splash screen and information to the screen. Now, Rex is ready to process ROS-based C/C++ source files and generate tuple-attribute models.

A.2 Using ClangEx

ClangEx is an interpretative command line tool that allows users to run multiple analysis jobs in a single run of the program. Figure A.1 shows the steps required to process C or C++ projects using ClangEx. In this figure, green boxes represent mandatory steps and purple boxes represent optional steps. In the pipeline, a user starts ClangEx, adds the C or C++ files they want analyzed, enables or disables certain language features to include in the final model, generates a model that represents those source files, and then outputs that model. As per the figure, multiple models can be generated and outputted in a single run of ClangEx.

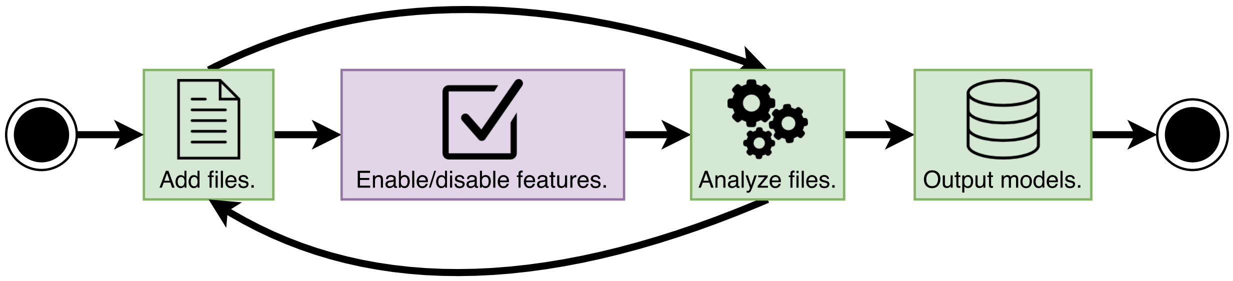


Figure A.1: The ClangEx processing pipeline.

When ClangEx is first started, it displays an empty prompt and waits for a user's commands. Table A.3 shows a list of common commands that ClangEx accepts and Table A.2 shows the commands for low-memory mode. To display help information, simply type `help`. For help on a specific ClangEx command, the command `help [COMMAND]` provides tailored help information for that specific command. The remainder of this guide will walk through the ClangEx processing pipeline shown in Figure A.1.

A.2.1 Step 1 - Adding and Removing Files

Before ClangEx is able to generate a model of the software project, files need to be added to the queue. This can be done using the `add` command. Removing files from the processing queue is also possible by using the `remove` command. There are two ways to add files to ClangEx: adding an individual file directly or recursively adding files in bulk using a directory. To add an individual file directly, use the `add -s [FILE]` command. In this command, the `[FILE]` portion represents the path to the file to add. Files added in this fashion can have any extension. If you attempt to add a file that does not yet exist, ClangEx will display a warning that this file does not exist and will ask if you want to continue. It is acceptable to have files in the processing queue that do not yet exist as long as they exist in step 3.

Files can be added in bulk through a built-in source file search tool. The command `add [DIRECTORY]` will do this. In this command, `[DIRECTORY]` represents a directory that contains a collection of source files you want added. This process is recursive; ClangEx will search all directories that are descendants to the directory passed in this add command. As an example, say the command `add /` is used.From this, ClangEx will add every single C or C++ source file on the system to the processing queue. Although C and C++ files can technically end with any extension, this search tool only looks for files that end with the following extensions: (1) .c; (2) .cc; and (3) .cpp.

Removing files is also possible. There are two ways files can be removed; individually by specifying their full path or by using a regular expression to match a collection of files. To remove a single file, use the `remove -s [FILE]` command. In it `[FILE]` is the absolute path of the file to be removed. If this does not match any files in the queue, ClangEx will display an error. To remove files in bulk, use the `remove -r [REGEX]` command. Here, `[REGEX]` is a regular expression that will be used to remove any file that matches it. Importantly, ClangEx will apply the regular expression on the **whole** file path. Once complete, ClangEx will report the number of files removed; this can be zero if the regular expression does not match any files.

A.2.2 Step 2 - Enabling or Disabling Features

An optional step of the processing pipeline is the ability to enable or disable C and C++ language features that are included in resultant TA models. By default, all language features are enabled when the program starts. If a language feature is enabled or disabled, it will remain that way until it is enabled or disabled again or until ClangEx is restarted. Table A.1 shows all the language features that can be enabled or disabled.

Language Feature Name	*Description*
cSubSystem	Directories that contain source code files.
cFile	Source code files (headers and source files).
cClass	C++ classes.
cFunction	Function definitions/declarations.
cVariable	Any type of variable (fields and variables).
cEnum	Enumerations and enumeration constants.
cStruct	Structure definitions.
cUnion	Union definitions.

Table A.1: ClangEx language feature names.

To enable a language feature simply use the `enable [FEATURES...]` command. In this command, the `[FEATURES...]` portion is a list of language features to enable separated by spaces. For instance, say a user wants to enable the *cVariable* and *cClass* features. To do this, they would type `enable cVariable cClass`. To disable these features, the `disable [FEATURES...]` command is used. This command works in the same fashion. If a user would like to disable the *cEnum* feature, simply type `disable cEnum`.

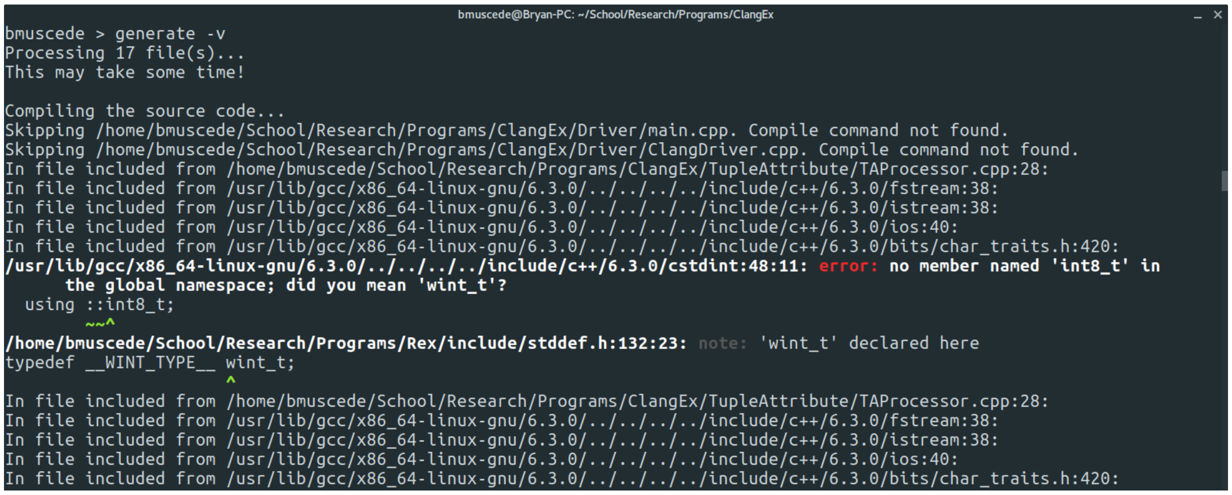
A.2.3 Step 3 - Analyzing Files

Once files have been added and language features have been *optionally* enabled or disabled, ClangEx can now process this source code to generate a model. The command to start this analysis is `generate`. Figure A.2 shows ClangEx while it is analyzing source code. Note that ClangEx will output any compiler errors or warnings that are encountered during analysis.

The `generate` command will only work if there is at least one file in the processing queue. If there are no files in the queue, ClangEx will display an error message.

With the generate command, two different processing modes can be set: *regular* and *full-analysis* mode. **By default**, ClangEx generates models using regular mode. This means that using the `generate` command with no arguments yields a model created using the regular analysis methodology. If a user wants to generate a model using the full-analysis methodology, they can use the following command: `generate -b`.

Using the two methodologies can result in drastically different models. Regular mode only analyzes the contents inside each source file and ignores **all** header files. Therefore, if a function is declared and defined inside a header file, regular mode will ignore it. The



```
bmuscede > generate -v
Processing 17 file(s)...
This may take some time!

Compiling the source code...
Skipping /home/bmuscede/School/Research/Programs/ClangEx/Driver/main.cpp. Compile command not found.
Skipping /home/bmuscede/School/Research/Programs/ClangEx/Driver/ClangDriver.cpp. Compile command not found.
In file included from /home/bmuscede/School/Research/Programs/ClangEx/TupleAttribute/TAProcessor.cpp:28:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/fstream:38:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/istream:38:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/ios:40:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/bits/char_traits.h:420:
/usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/cstdint:48:11: error: no member named 'int8_t' in
      the global namespace; did you mean 'wint_t'?
  using ::int8_t;
        ~~^
/home/bmuscede/School/Research/Programs/Rex/include/stddef.h:132:23: note: 'wint_t' declared here
typedef __WINT_TYPE__ wint_t;
                      ^
In file included from /home/bmuscede/School/Research/Programs/ClangEx/TupleAttribute/TAProcessor.cpp:28:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/fstream:38:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/istream:38:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/ios:40:
In file included from /usr/lib/gcc/x86_64-linux-gnu/6.3.0/../../../../include/c++/6.3.0/bits/char_traits.h:420:
```

Figure A.2: ClangEx while running on some unspecified source code.

full-analysis methodology does the opposite; it looks at each source file and all header files that file includes and processes the contents (minus system header files).

In addition, since analysis typically results in an in-memory hierarchical graph being generated on the fly, some large projects can cause ClangEx to run out of memory. To combat this, the `--low` flag can be used when invoking the `generate` command. This causes ClangEx to dump the hierarchical graph to disk several times per source file. By default, ClangEx dumps these files in the root directory of ClangEx but the location can be changed by using the `outLoc [DIRECTORY]` command before invoking `generate`. Low-memory mode is less likely to crash but is slower since it requires disk usage.

In the event ClangEx crashes or fails during the `generate` step for **low-memory** mode, a graph can be recovered using the `recover` command. Recover will resume processing a run of low-memory TA generation by examining the files dumped to disk. By default, `recover` will try to resume processing the code where ClangEx left off but `recover -c` will tell ClangEx to finish the graph as-is and save it to disk as a full-fledged TA model. If the `outLoc` command is used in the previous run of ClangEx, the `recover -i [DIRECTORY]` can be used to point ClangEx to where the previous graph files are located.

One last caveat with generating models. If the project being processed uses any compiler flags, a compilation database will need to be generated for that project an placed in the root directory of all the project source files. All files being analyzed by ClangEx must either be in the same directory as the database or in some descendant directory. A compilation database is a JSON file that has an entry for **each** file that ClangEx will analyze. If a file is not present in that database, it will be ignored. If a compilation database does not exist, ClangEx will show a warning and attempt to proceed without it.

A.2.4 Step 4 - Outputting Models

Once ClangEx has generated models, they need to be outputted to disk as TA files. As ClangEx can generate multiple models in one program run, the output command is designed to output multiple models at once. This step is divided into two parts. There is a guide for users who only want to output a single model and a guide for users who want to output multiple models.

One Model - If only one model was generated in a ClangEx run, outputting that model is extremely easy. The command `output [FILENAME]` can be used. In this command, the `[FILENAME]` portion represents the name of TA model to output. This filename **should not** include an extension at the end; ClangEx will automatically add that for you in the form of the `.ta` extension.

Multiple Models - There are two options to output multiple models using ClangEx. They can either be all outputted at the same time or a specific model can be outputted by typing its model number. To output all models at once, use the `output [BASE_FILENAME]` command. In this case, `[BASE_FILENAME]` is a base file name that will be shared by all models. Each model will be saved with this base name with its model number and extension appended to the end. For instance, say a user types `output baseName` and there are two models to output. In that case, ClangEx will output two files: baseName-0.ta and baseName-1.ta. The other method allows users to output a specific model. To do this, the command `output -s [NUM] [FILENAME]` needs to be used. In this command, the `[NUM]` option specifies the model number. For each model, its model number is displayed when that model is generated by the `generate` command.

A.3 Using Rex

As Rex is based upon ClangEx, it operates in a similar manner. Rex's processing pipeline is the exact same as ClangEx with two differences. First, users cannot enable or disable specific language features with Rex. Second, there is an optional step after file analysis called *resolve features*. Figure A.3 shows the Rex pipeline in detail. In this, green boxes represent mandatory steps and purple boxes are optional steps. This pipeline is capable of generating multiple models in a single run of the Rex extractor. There are four major steps when analyzing projects using Rex: (1) adding files to the processing queue; (2) building a model of all source files in the processing queue; (3) resolving which classes belong to which features; and (4) outputting all generated models to disk as tuple-attribute files.

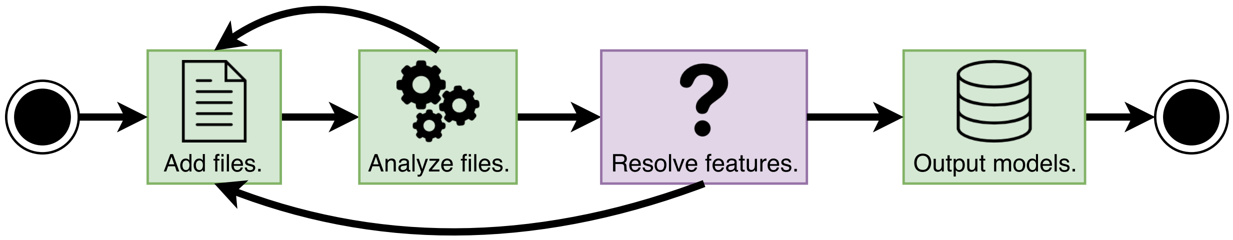


Figure A.3: The Rex processing pipeline.

Because the Rex pipeline is almost identical to the ClangEx pipeline, it uses the same commands as ClangEx. Table A.4 gives common Rex commands that can be used while running the extractor and Table A.2 gives commands for low-memory mode. This section will not delve into too much detail on how to use Rex. For an example on how to add files, analyze files, or output models, refer to the ClangEx guide. The remainder of this guide will describe the *resolve features* step and discuss any other differences between Rex and ClangEx.

A.3.1 Optional Step - Feature Resolution

Once a project is analyzed using Rex, feature resolution can be conducted. This is an optional step that can be used when entire ROS projects are analyzed at once. For instance, say a user is analyzing a ROS project with four different ROS packages (features). The feature resolution step links each class and ROS component with the ROS package that it is declared in.

To perform feature resolution, the command `resolve [DIRECTORY]` can be used. It can only be used once a ROS project has been analyzed. The `[DIRECTORY]` portion of the command specifies a directory that contains a compilation database for each ROS package in the project. These compilation databases can be in a descendant directory of the directory supplied.

Low-Memory Mode Reference Sheet		
Command	*Usage*	*Description*
generate	`generate -l`	Generates a TA model using the low-memory analysis methodology. Dumps a model to disk for every file processed.

recover	`recover -c`
	`recover -i [DIRECTORY]`
		Recovers a TA model from a previous run of low-memory analysis. The `-i` flag specifies the directory the previous files were located in and the `-c` flag just generates a TA model from those intermediate files.

outLoc	`outLoc [DIRECTORY]`	Allows a user to change the directory the low-memory mode files are outputted to.

Table A.2: An overview of low-memory mode commands for ClangEx and Rex.

A.3.2 Differences Between Rex and ClangEx

There are two other notable differences between Rex and ClangEx from an operational perspective. The first is that Rex has no enable/disable language feature. This means that all language features present in the TA metamodel will always be included. Although Rex does not have this functionality, nodes or edges in the TA models can be retroactively removed through Grok scripts. Additionally, unlike ClangEx, all language features in the Rex metamodel are required for effective hotspot detection.

The other major difference is that, instead of ClangEx's regular and full-analysis modes, Rex has the simple-analysis and full-analysis processing modes. The models that these two modes generate are at different levels of granularity. Simple-analysis mode generates a model that shows which classes communicate with which whereas full-analysis mode generates a heavier model showing all C++ language features. Full-analysis mode is enabled by default and is used when the `generate` command is typed. Simple-analysis mode needs to be activated by using the `-m` flag with the generate command: `generate -m`.

ClangEx Quick Reference Sheet		
Command	*Usage*	*Description*
help	`help [COMMAND]`	Displays program help information. Can also display specific information tailored to a particular command.

add	`add [DIRECTORY]`
	`add -s [FILE]`
		Adds a single source file (with the `-s` flag) or recursively adds all the source files inside a directory. These will be processed by ClangEx.

remove	`remove -r [REGEX]`
	`remove -s [FILE]`
		Removes all files that match a certain regular expression (with the `-r` flag) or a singular file name (with the `-s` flag).

enable	`enable [FEATURES...]`	Allows ClangEx to process specific language features. See Section A.2.2 for a list of language features that can be turned on.
disable	`disable [FEATURES...]`	Stops ClangEx from processing specific language features. See Section A.2.2 for a list of language features that can be turned off.

generate	`generate`
	`generate -b`
		Processes all source files that have been added using the `add` command. The `-b` flag is for *full-analysis* mode.

output	`output [FILENAME]`
	`output -s [NUM] [FILENAME]`
		Outputs all models stored by ClangEx. By specifying a file name, ClangEx will output all TA models using that file name as a base. Also, the `-s` flag outputs an individual model based on its model number.

script	`script [SCRIPT_FILE]`	Runs a specified script file. Scripts are a collection of ClangEx commands in some text file. When the script finishes, ClangEx will return back to its command prompt.

Table A.3: An overview of common ClangEx commands.

Rex Quick Reference Sheet		
Command	*Usage*	*Description*
help	`help [COMMAND]`	Displays program help information. Can also display specific information tailored to a particular command.

add	`add [DIRECTORY]`
	`add -s [FILE]`
		Adds a single source file (with the `-s` flag) or recursively adds all the source files inside a directory. These will be processed by Rex.

remove	`remove -r [REGEX]`
	`remove -s [FILE]`
		Removes all files that match a certain regular expression (with the `-r` flag) or a singular file name (with the `-s` flag).

generate	`generate`
	`generate -m`
		Processes all source files that have been added using the `add` command. The `-m` flag tells Rex to generate the model in simple-analysis mode.

resolve	`resolve [DIRECTORY]`	Identifies which ROS package each class and ROS component belongs to. The directory specified must have a separate compilation database for each ROS package.

output	`output [FILENAME]`
	`output -s [NUM] [FILENAME]`
		Outputs all models stored by Rex. By specifying a file name, Rex will output all TA models using that file name as a base. Also, the `-s` flag outputs an individual model based on its model number.

script	`script [SCRIPT_FILE]`	Runs a specified script file. Scripts are a collection of Rex commands in some text file. When the script finishes, Rex will return back to its command prompt.

Table A.4: An overview of common Rex commands.

Appendix B

Installing & Using the bfx64 Extractor

bfx64 is a fact extractor designed to collect basic program information from object files. Based upon Jingwei Wu's *BFX* extractor [38], bfx64 is capable of processing 32-bit and 64-bit ELF-based object files to generate a TA model that contains information about that object file's functions, variables, and the relationships amongst them. Although this fact extractor was not covered in this thesis, bfx64 was developed during this thesis.

B.1 Installing bfx64

Before bfx64 can be installed, there are several libraries and tools that must be present on the target system. First, because ELF object files are encoded in a binary format, bfx64 uses a custom C++ library called **ELFIO** to assist with object file processing. In addition, **CMake** is required to build bfx64 and the C++ **Boost** libraries are required for command line parsing and file processing. Once each of these are installed, bfx64 can be built from source and run.

The remainder of this section walks through installing CMake, Boost, and ELFIO on a target Linux system and then shows how bfx64 can be built from source. Section B.1.1 describes CMake, Boost, and ELFIO installation. Section B.1.2 describes how to build bfx64.

B.1.1 Prerequisites

CMake

If using a Ubuntu or Debian-based system, installing CMake is as simple as using `apt-get`. This is done using the following two commands:

```
$ sudo apt-get install cmake
$ cmake --version
```

Using `apt-get` might not install the latest version of CMake. However, as long as it installs a version of CMake greater than `3.0.0`, it can be used to build bfx64. If this is the case, proceed to the Boost installation instructions.

If this method did not work, CMake must be built from source. To do this, the latest version of CMake source needs to be downloaded from the CMake website, compiled, and then installed. This guide provides instructions on how to build CMake `3.7.0` from source. The first step is to download the CMake source code and unzip it. This can be from the command line using the following commands:

```
$ wget https://cmake.org/files/v3.7/cmake-3.7.0.tar.gz
$ tar xvzf cmake-3.7.0.tar.gz
$ cd cmake-3.7.0
```

Once in the `cmake-3.7.0` directory, CMake can be configured and install on the target system. This process may take several minutes. To do this, use the following commands:

```
$ ./configure
$ make
$ make install
$ cmake --version
```

If these steps completed successfully, CMake is now installed and ready for use.

Boost

Installing Boost on Ubuntu or Debian-based systems is extremely easy by using the `apt-get` package manager. This installation process will install the libraries and add them to the target system's default *include* path. This can be done using the following command:

```
$ sudo apt install libboost-all-dev
```

ELFIO

To install ELFIO, it needs to be downloaded and then installed to the target system's default *include* path. As ELFIO is not a part of the Ubuntu or Debian software universe, manual installation is required. First, the source code needs to be downloaded from the ELFIO website and extracted. This can be achieved using the following command:

```
$ wget https://downloads.sourceforge.net/project \
        /elfio/ELFIO-sources/ELFIO-3.2/elfio-3.2.zip
$ gunzip elfio-3.2.zip
$ cd elfio-3.2
```

With ELFIO downloaded and unzipped, it can now be installed. To do this, the following shell script needs to be run:

```
$ ./install-sh.sh
```

B.1.2 Building bfx64

To build bfx64, the source code needs to be checked out from the bfx64 GitHub repository and then built using CMake. If bfx64 fails to build, first ensure that all tools and libraries installed in the previous section are present on the target system.

To download bfx64, first checkout the source code from GitHub by using the following command:

```
$ git checkout https://github.com/bmuscede/bfx64.git
```

If this completed successfully, a directory called `bfx64` will be created that contains all the bfx64 source code.

The next step is to build bfx64. These steps install it in a directory called `bfx64-Build` adjacent to the `bfx64` directory. To install bfx64 somewhere else, simply replace the `bfx64-Build` string in the following commands with a desired directory name. The following steps install bfx64 in the `bfx64-Build` directory:

```
$ mkdir bfx64-Build
$ cd bfx64-Build
$ cmake -G "Unix Makefiles" ../bfx64
$ make
```

By running these commands, CMake will build the source code and an executable called *bfx64* will be created inside the `bfx64-Build` directory. To verify that bfx64 built correctly, run:

```
$ ./bfx64 --help
```

This will show bfx64's version and license. Now, bfx64 is ready to process ELF-based, C/C++ object files and generate TA models.

B.2 Using bfx64

As previously stated, bfx64 is used to analyze ELF-based object files that correspond to C or C++ source code. Using bfx64 is easy as it runs on the command line, automatically finds and processes object files, and then outputs a resultant TA model.

The remainder of this section describes how to use bfx64. Section B.2.1 highlights basic commands and Section B.2.2 highlights advanced commands. For more information about a command, refer to bfx64's help screen by running `bfx64 --help`.

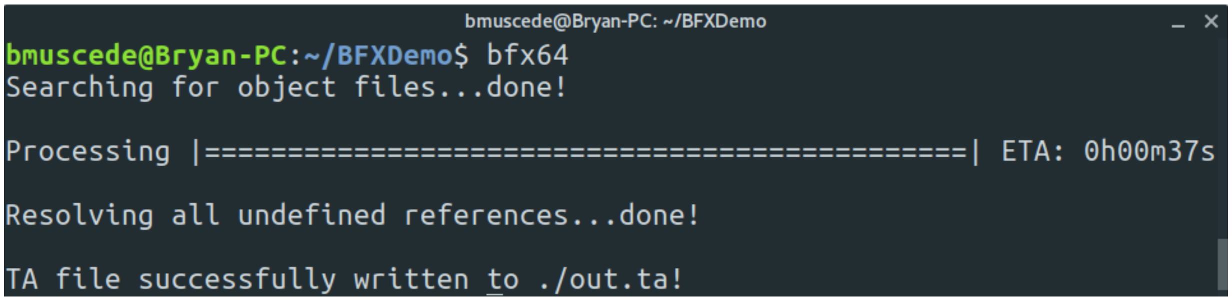
B.2.1 Basic Usage

The easiest way to generate a TA model of object files is to run bfx64 with no arguments. In the terminal, navigate to the root directory of a project to analyze and then type `bfx64`. By doing this, bfx64 will scan the current directory and all descendant directories for object files. These files will then be processed one at a time. Lastly, bfx64 will save the resultant TA model to disk. By default, the TA model is saved to `./out.ta`. Figure B.1 shows what bfx64 looks like as it is processing object files inside a demo directory.

Basic Command Line Arguments

bfx64 has four basic command line arguments that can be set to alter its analysis. Table B.1 provides quick reference information for each of these arguments. The remainder of this section will describe each of these arguments in detail.

The first argument is `--output (-o) [NAME]`. This allows users to specify where the resultant TA model is outputted to. The `[NAME]` portion of the argument should be a relative or absolute path. As shown in B.1, if the `--output` argument is not used, the TA model will be saved to `./out.ta`.



```
bmuscede@Bryan-PC: ~/BFXDemo
bmuscede@Bryan-PC:~/BFXDemo$ bfx64
Searching for object files...done!

Processing |==============================================| ETA: 0h00m37s

Resolving all undefined references...done!

TA file successfully written to ./out.ta!
```

Figure B.1: A run of bfx64 with no arguments.

The `--dir (-d) [DIRECTORY]` argument can be used to change where bfx64 looks for object files. If this argument is not used, bfx64 will just look in the current working directory for object files. The `[DIRECTORY]` portion of the argument is the directory for bfx64 to look in.

The `--verbose (-v)` flag causes bfx64 to print more detailed information to screen. Instead of showing a progress bar while bfx64 is processing object files (as shown in Figure B.1), verbose mode prints information about each object file as it is being processed.

Last, the `--help (-h)` flag displays license information and program options. This flag is processed prior to evaluating any other program argument.

Basic bfx64 Program Arguments		
Long Option	*Short Option*	*Description*
`--output [NAME]`	`-o [NAME]`	Changes the name of the file that bfx64 outputs.
`--dir [DIRECTORY]`	`-d [DIRECTORY]`	Changes the directory where bfx64 searches for object files.
`--verbose`	`-v`	Shows verbose processing information.
`--help`	`-h`	Shows program help information.

Table B.1: A collection of basic bfx64 arguments.

B.2.2 Advanced Features

There are also some advanced arguments that provide more fine-grained control over bfx64. Table B.2 shows the advanced program arguments that bfx64 supports. There are two types of advanced features: *file processing arguments* and *low-memory arguments*.

Advanced bfx64 Program Arguments		
Long Option	*Short Option*	*Description*
`--suppress`	`-s`	Forces bfx64 to not search for object files.
`--object [FILENAME]`	`-i [FILENAME]`	Allows users to specify an object file to include.
`--exclude [FILENAME]`	`-e [FILENAME]`	Allows users to specify an object file to exclude.
`--low`	`-l`	(Low-memory mode) Dumps the in-memory graph to disk every so often to prevent large projects fro crashing bfx64.
`--dump [NUM]`	`-u [NUM]`	(Low memory mode) Sets the interval in which the graph is dumped to disk.

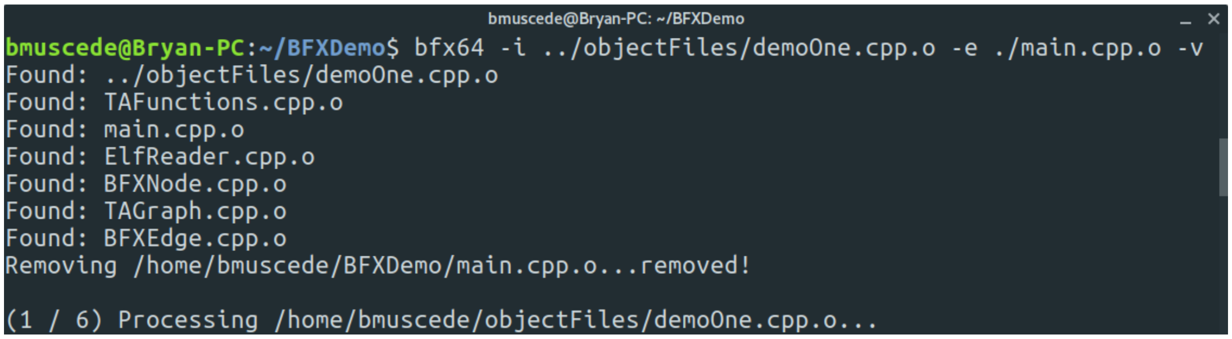
Table B.2: A collection of advanced bfx64 arguments.

File Processing Arguments

There are three file processing arguments that can be used. First, users can manually specify files they want to include or exclude from processing. The `--object (-i) [FILENAME]` argument allows users to specify files they want to add. The `--exclude (-e) [FILENAME]` argument allows users to remove files from the processing queue. Both these options can be used multiple times. The `--object` argument is useful for adding extra files in conjunction to bfx64's search tool. The `--exclude` argument is useful for removing specific object files from the object files found using bfx64's search tool. For example, if a project has hundreds of object files in a single directory and a user wants to process all but one of these object files, they can simply exclude that one object file.

The other file processing argument is the `--suppress (-s)` flag. It forces bfx64 to not search for object files and, instead, rely only on object files added using the `--object` option. If this flag is used, the `--object` option must also be used at least once or else bfx64 will have no object files to process.

As an example of these file processing arguments, say a user wants to process a series of object files inside a directory called `bfx64Demo` as well as an additional object file called `demoOne.cpp.o` inside an adjacent directory called `objectFiles`. Also, although that user wants to process most of the object files inside `bfx64Demo`, they do not want to process the



```
bmuscede@Bryan-PC: ~/BFXDemo
bmuscede@Bryan-PC:~/BFXDemo$ bfx64 -i ../objectFiles/demoOne.cpp.o -e ./main.cpp.o -v
Found: ../objectFiles/demoOne.cpp.o
Found: TAFunctions.cpp.o
Found: main.cpp.o
Found: ElfReader.cpp.o
Found: BFXNode.cpp.o
Found: TAGraph.cpp.o
Found: BFXEdge.cpp.o
Removing /home/bmuscede/BFXDemo/main.cpp.o...removed!

(1 / 6) Processing /home/bmuscede/objectFiles/demoOne.cpp.o...
```

Figure B.2: A run of bfx64 with some advanced processing options enabled.

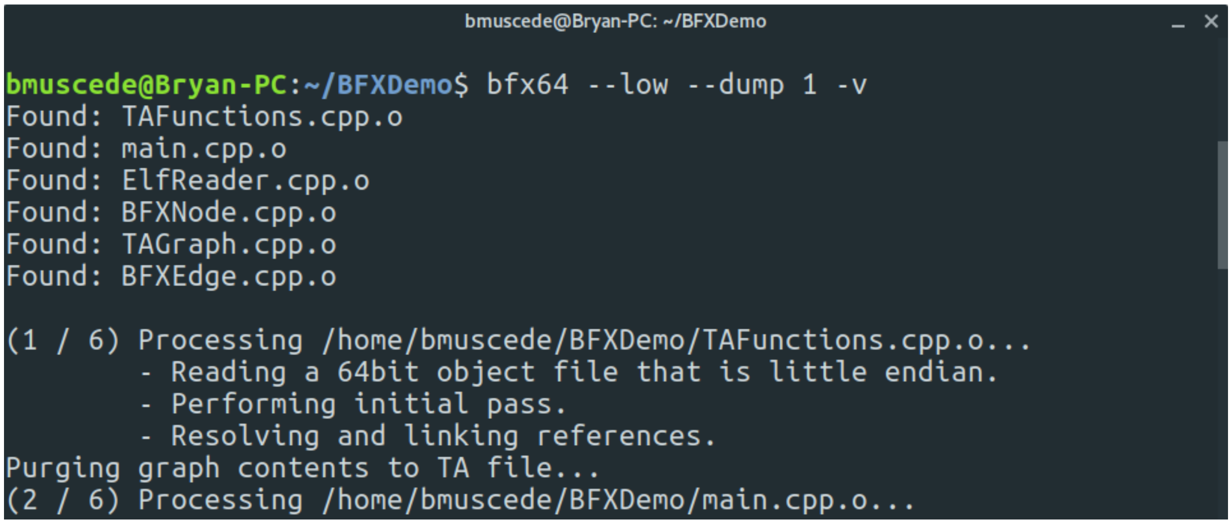
`main.cpp.o` object file. Figure B.2 shows how this can be achieved using bfx64. In this example the `-i` argument is used to tell bfx64 to include the `demoOne.cpp.o` object file and the `-e` argument is used to tell bfx64 to exclude the `main.cpp.o` object file. Lastly, the `-v` flag puts bfx64 into verbose mode. This tells the user which files have been found and which files have been added and removed.

Low-Memory Arguments

Because some software projects can consist of thousands of object files, bfx64 has two arguments that improve performance on low-memory systems. The `--low (-l)` flag tells bfx64 to enable low-memory mode. In this mode, bfx64 will dump its in-memory TA graph to disk every so often to free up memory. By default, this graph dump occurs every 100 object files. The `--dump (-u) [NUM]` argument allows users to set their own dump interval. Because this argument can only be used in low memory mode, using it without the `--low` flag will cause bfx64 to display an error.

As an example, Figure B.3 shows a run of bfx64 in low-memory mode. Here, low-memory mode was enabled by using the `--low` flag and bfx64 was set to dump the TA model after every object file is processed[1]. Lastly, verbose mode is set using the `-v` flag. When low-memory mode is enabled, verbose mode notifies the user every time the object file is dumped.

[1]This is merely for demonstration. Because dumping the model impacts performance, a user would want to set this value as high as possible without causing bfx64 to crash.



```
bmuscede@Bryan-PC:~/BFXDemo$ bfx64 --low --dump 1 -v
Found: TAFunctions.cpp.o
Found: main.cpp.o
Found: ElfReader.cpp.o
Found: BFXNode.cpp.o
Found: TAGraph.cpp.o
Found: BFXEdge.cpp.o

(1 / 6) Processing /home/bmuscede/BFXDemo/TAFunctions.cpp.o...
        - Reading a 64bit object file that is little endian.
        - Performing initial pass.
        - Resolving and linking references.
Purging graph contents to TA file...
(2 / 6) Processing /home/bmuscede/BFXDemo/main.cpp.o...
```

Figure B.3: A run of bfx64 in low-memory mode.

Appendix C

Relational Algebra Scripts

This chapter presents all Grok relational algebra scripts that were used in the case study on Autonomoose. There is a script for each of the seven types of hotspots that were presented in this thesis and can be used for programs that utilize ROS for communication between features. These scripts were each written to operate on TA models generated by the Rex extractor and utilize the jGrok syntax[1]. The remainder of this chapter presents each of the hotspot scripts and describes how they work.

C.1 Feature-Communication Hotspots

C.1.1 Inter-Component-Based Communication

Figure C.1 shows the Grok script that detects instances of the inter-component-based communication hotspot. In it, lines 1 through 6 check arguments given to the script to ensure that a TA model was supplied. Lines 9 through 11 initialize Grok by setting the `$INSTANCE` variable to the empty set and then by loading the desired TA model into memory.

Then, line 14 of Figure C.1 gets all direct communications between components by joining the publish and subscribe relations using the selection (`o`) operator and then by lifts that joined relation up to the parent class. Line 17 through 30 gets the indirect instances of communication. To do this, the publish and subscribe relations are joined again in line

[1]jGrok syntax can be accessed here: http://www.swag.uwaterloo.ca/jgrok/grokdoc/operators.html

```
 1 //Argument check.
 2 if ($# != 1) {
 3   print "Usage: Grok1.ql [ROS_MODEL]"
 4   print "Gets the features that directly/indirectly call each other."
 5   return;
 6 }
 7
 8 //Sets up Grok.
 9 $INSTANCE = eset;
10 inputFile = $1;
11 getta(inputFile);
12
13 //Gets the direct communications
14 direct = contain o (publish o subscribe) o (inv contain);
15
16 //Combines publishers and subscribers with function calls.
17 rosComm = publish o subscribe;
18 fullCall = rosComm + call;
19 fullCall = fullCall+;
20
21 //Gets a list of publishers and subscribers.
22 publishSet = $INSTANCE . {"rosPublisher"};
23 subscribeSet = $INSTANCE . {"rosSubscriber"};
24
25 //Ensures publishers start and subscribers end.
26 comms = publishSet o fullCall o subscribeSet;
27
28 //Gets the indirect communication.
29 indirect = contain o comms o inv contain;
30 indirect = indirect - direct;
31
32 //Prints the results.
33 print "Direct Messages:";
34 inv @label o (compContain o direct o inv compContain) o @label;
35
36 print "Indirect Messages:";
37 inv @label o (compContain o indirect o inv compContain) o @label;
```

Figure C.1: Grok script that detects the inter-component-based communication hotspot.

17 and then combined with the function call relation using the union operator (`+`) into a set called `rosComm`. This set has every function call or instance of communications between features in the project. Line 19 gets the transitive closure of the `fullCall` relation. With

the transitive closure of `fullCall` generated, lines 22 and 23 create a set of publisher objects (`publishSet`) and subscriber objects (`subscribeSet`). This is done by taking the set of entities in the model and only selecting any publisher or subscriber objects. The purpose of the `publishSet` and `subscribeSet` is that, in line 26, the script filters out results from `rosComm` that do not start with a publisher and end with a subscriber. This gives us all the inter-component-based communications between features. Last, line 29 lifts the indirect relation to the class level and line 30 removes and direct results. The resulting values are then printed in lines 33 through 37.

C.1.2 Loop Detection

Figure C.2 shows the Grok script that detects any loops in the feature communication graph. Again, for brevity, lines 1 through 11 of the inter-component-based communication script in Figure C.1 were omitted from this script. From a high-level perspective, this script is similar to the first script in getting direct and indirect communications. The difference here is that this script removes any entries from the direct and indirect relations that do not have the same domain and range.

Line 2 in Figure C.2 generates a relation using the identity operator where, for each class in the model, there is an entry where the domain and range is itself. This means that, in this relation, each class in the model points to itself. Although it may appear useless, this `loopTest` relation is used later to determine whether there is a direct or indirect loop. Like in lines 17 through 19 of Figure C.1, lines 5 and 6 combine the relation of messages between functions and function calls and then get the transitive closure. This is stored in a relation called `fullCall` and will be used in later lines to get the indirect loops.

Lines 9 and 10 get the direct loops in the model by getting all the features that communicate with other features. This is done by joining the `publish` and `subscribe` relations by topic and then lifting to the class level. Because this gets all communications and not just loops, to remove any non-loops, the intersection operator (`^`) is used on the `classComm` relation and `loopTest` relation. This means only entries in the `classComm` relation that has the same domain and range are kept. These are the direct loops.

Lines 13 through 19 get the indirect loops. This is achieved by taking the `fullCall` set (created in lines 5 and 6), ensuring that it only starts with a publisher and ends with a subscriber, and then removing any non-loops. Lastly, lines ww through 26 print the results of each.

```
//Generates a list of classes pointing back to themselves.
loopTest = id ($INSTANCE . {"cClass"});

//Generates the call graph.
fullCall = (publish o subscribe) + call;
fullCall = fullCall+;

//Generates the direct loops.
classComm = contain o (publish o subscribe) o (inv contain);
direct = classComm ^ loopTest;

//Gets a list of publishers and subscribers.
publishSet = $INSTANCE . {"rosPublisher"};
subscribeSet = $INSTANCE . {"rosSubscriber"};

//Gets the dataflow indirect results.
comm = contain o publishSet o fullCall o subscribeSet o inv contain;
indirect = comm ^ loopTest;
indirect = indirect - direct;

//Prints the results.
print "Direct Loops:";
inv @label o (compContain o direct o inv compContain) o @label;

print "Indirect Loops:"
inv @label o (compContain o indirect o inv compContain) o @label;
```

Figure C.2: Grok script that detects the loop hotspot.

C.2 Multiple Publisher Hotspots

C.2.1 Multiple Input

Figure C.3 shows the Grok script that detects the multiple input hotspot. For brevity, lines 1 through 11 of the inter-component-based communication script in Figure C.1 were omitted from this figure. Additionally, to further simplify this script, a built-in Grok function called `indegree` was used to count the number of publishers or topics being received.

The script starts by generating a relation called `direct` in lines 2 and 3 by joining the `publish` and `subscribe` relations together by topic and then by getting the parent class that sends messages to some subscriber. This is done by joining `contain` with `direct`

```
 1 //Gets subscribers that are written to.
 2 direct = publish o subscribe;
 3 direct = contain o direct;
 4 
 5 //Gets the indegree
 6 inset = indegree(direct);
 7 
 8 //Removes items from indegree.
 9 inset = inset [ &1 > 1 ];
10 
11 //Prints the results.
12 print "Subscribers that have Multiple Component Communications:";
13 inv @label o (compContain . dom inset);
14 
15 //Now, dives deeper.
16 for item in dom(inset) {
17   cbFunc = {item} . call;
18   cFunction = (rng(cbFunc o @label));
19   cClass = rng((dom(contain o rng{item})) o @label);
20 
21   //Print the items.
22   print "Callback Function " + cFunction + " (" + cClass + ") - ";
23 
24   //Get the publishers.
25   inv @label o (compContain . (direct . {item}));
26 }
```

Figure C.3: Grok script that detects the multiple publishers hotspot.

on the domain of the `direct` relation. By doing this, the `direct` relation has record of situations where multiple publishers message the same topic and where multiple topics message the same subscriber. Next, line 6 uses the `indegree` function on the `direct` relation. This function looks at the range of the `direct` relation and counts the number of times each entity ID in the range appears. As output, it generates a relation of the form `<SUBSCRIBER_ID> <NUM_OCCURRENCES>`. Then, because this script is only interested in subscribers that receive messages from numerous features, this script filters out any entries in `inset` relation where the number of occurrences is only 1 or less. To do this, line 9 takes the `inset` relation and checks if the range has a value of one or less. If it does, it removes that entry from `inset`.

Once the `inset` relation has been whittled down to remove any subscribers that receive messages from only one feature, this relation now contains all multiple publisher hotspots in the project. Lines 12 and 13 print those results. If the `inset` relation is empty, it means there are no instances of this hotspot.

The last part of the script provides more detail about each multiple publisher instance detected. The script loops through the domain of the `inset` relation (the ID of the subscriber) and then, for each, lines 17 through 18 get the callback function the subscriber messages and the class the subscriber resides in. Getting the callback function is achieved by joining the subscriber to the `call` relation using the `.` operator. This works because the subscriber messages the callback function via the `call` relation when data is received. Getting the class in line 19 involves taking the `contains` relation and joining the subscriber to the range of that relation. This gets the parent class. Line 22 prints the function and class the subscriber is in and then line 25 prints all the features that message that subscriber.

C.2.2 Race Condition

Figure C.4 shows the Grok script that is used to detect the race condition hotspot. Although the process to detect this hotspot might appear to be complex, it is actually fairly easy. Like all previous scripts, for brevity, lines 1 through 11 of the inter-component-based communication script in Figure C.1 were omitted from this figure.

First, line 2 gets a list of callback functions in the project by taking the set of ROS subscribers and joining it with `call` relation. Because this creates a relation of the form `<SUBSCRIBER> <CALLBACK_FUNCTION>`, the range operator (`rng`) is applied to that new relation to get a set of callback functions. Then, line 5 gets all the variables written to by these callback functions. This is done by taking the `callbackFunc` set and using the selection operator (`o`) on the domain of the `write` relation to generate a relation of the form `<CALLBACK_FUNCTION> <VARIABLE>`. Line 6 uses the `indegree` function on the `vars` relation. This function looks at the range of the `vars` relation and counts the number of times each variable is written to. As output, it generates a relation of the form `<VARIABLE>` `<NUM_OCCURRENCES>`. Last, line 9 removes entries from the range of `inset` to keep only entries that have a range of two or greater.

Lines 12 through 21 contain a `for` loop block that loops through every variable in the `inset` relation created in the previous lines. Because the range operator returns a set with no duplicates as per relational algebra, each variable written to will only be processed once by this loop. Line 13 takes the current variable and applies it to the `vars` relation to get all cases where that variable is written to. Lines 16 through 20 prints information about these variables. If a variable is written to by more than one callback function, the name

```
 1 //Get a list of callback functions.
 2 callbackFunc = rng (subscribe o call);
 3
 4 //Determines all the variables that are modified by each callback function.
 5 vars = callbackFunc o write;
 6 inset = indegree(vars);
 7
 8 //Purges variables written to by less than 2 functions.
 9 inset = inset [ &1 > 1 ];
10
11 //Gets more information about the relation.
12 for curVar in dom inset {
13   specific = vars . {curVar};
14
15   //Deal with cases where multiple callbacks modify.
16   print "For the " + curVar + " variable:";
17
18   //Gets the callback functions that push to that variable.
19   callbacks = dom specific;
20   callbacks . @label;
21 }
```

Figure C.4: Grok script that detects the race condition hotspot.

of the variable is printed in line 16. Then lines 19 and 20 gets all callback functions that modify this variable and prints each to the screen.

C.3 Control Flow Hotspots

C.3.1 Behaviour Alteration

Figure C.5 shows the Grok script that detects the behaviour alteration hotspot. For brevity, some portions of this script were removed or altered. This includes the initial argument check shown in the inter-component-based communication script in Figure C.1. From a high-level perspective, this script detects callback functions that write to variables that "eventually" participate in the decision portion of a control structure.

To carry out this detection, lines 2, 3, and 4 get instances of direct and indirect communication between features. Line 2 gets instances of direct communication between a feature and a recipient feature's callback function. To do this, the publish and subscribe relations

```
//Gets the direct and indirect relations.
direct = contain o publish o subscribe o call;
indirect = contain o publish o subscribe o inv contain;
indirect = indirect+;

//Generates relations to track the flow of data.
callbackFuncs = rng(subscribe o call);
controlFlowVars = @isControlFlow . {"\"1\""};
masterRel = varWrite + call + write + varInfFunc;
masterRel = masterRel+;

//Gets the behaviour alterations.
behAlter = callbackFuncs o masterRel o controlFlowVars;
print "There are " + #behAlter + " cases of behaviour alteration.";
print "Across " + #(dom behAlter) + " callback functions.";

//Loops through each callback function.
for item in dom behAlter {
  //Gets the callback function name.
  {item} . @label;
  print "";

  //Print the variables it affects.
  print "Affects Variables:"
  inv @label . ({item} . behAlter);
  print "";

  //Print features that directly influence.
  print "Influenced By - Direct:"
  dirInf = direct . {item};
  inv @label . dirInf;

  //Print features that indirectly influence.
  print "Influenced By - Indirect:";
  inInf = (indirect . (direct . {item})) - dirInf;
  inv @label . inInf;
  print "";
}
```

Figure C.5: Grok script that detects the behaviour alteration hotspot.

are joined by topic and then joined with the call relation to get the callback function that receives this data. Lastly, this is joined with contain relation on the left-hand side to get the parent class that sends the data. Line 3 does the same thing as line 2 except it lifts both ends of the relation up to the class level. Then line 4 gets the transitive closure of `indirect` to get all indirect instances of communication.

Now, the control flow variables and callback functions need to be obtained. Line 7 gets all callback functions by joining the subscribe relation with the call relation and then gets the range of that result. Line 8 gets all variables that participate in the decision portion of a control structure by taking the `isControlFlow` attribute and only keeping entries that have a value of 1. Lines 9 and 10 takes a collection of relations and combines them together. This includes the `varWrite`, `call`, `write`, and `varInfFunc` relations. Then, in line 10, the transitive closure of this relation is obtained. This gives a relation called `masterRel` that contains the dataflow of all variables and functions in the program.

Once the `masterRel` relation has been created, the behaviour alteration instances can be obtained. Line 13 does this by whittling down the masterRel relation to only start with callback functions and to only end with control flow variables. By doing this, it shows any callback functions that *eventually* modify variables that participate in the decision portion of control structures. The results are printed in lines 14 and 15.

Because printing the number of instances is not enough, lines 18 through 38 loop through each callback function in the set of behaviour alteration instances and print important details. In this loop, line 20 prints the name of the callback function being investigated. Then lines 24 and 25 print all the control flow variables that are affected by this callback function. This is done by taking the callback function being investigated and projecting it upon the `behAlter` relation.

Once this is complete, the direct and indirect features that influence this callback function are printed. Lines 29 through 31 print the direct instances. This is done by taking the direct relations and projecting it on the callback function. This gives all the features that write to that callback function. Lines 34 through 37 print the indirect instances. This is done by getting all direct instances and projecting them with the indirect relation.

C.3.2 Publish Alteration

Figure C.6 shows the Grok script that detects the publish alteration hotspot. For brevity, some portions of this script were removed or altered. This includes the initial argument check shown in the inter-component-based communication script in Figure C.1. Further, because this script is a slight modification of the behaviour alteration script, there are some similarities in how this hotspot is detected.

```
//Gets the direct and indirect relations.
direct = contain o publish o subscribe o call;
indirect = contain o publish o subscribe o inv contain;
indirect = indirect+;

//Generates relations to track the flow of data.
callbackFuncs = rng(subscribe o call);
masterRel = varWrite + varInfluence + varInfFunc + call + write;
masterRel = masterRel+;

//Gets publisher alterations.
pubAlter = callbackFuncs o masterRel o publish;
print "There are " + #pubAlter + " cases of publisher alteration.";

//Loops through each callback function.
for item in dom pubAlter {
  //Prints the name of the callback.
  {item} . @label;
  print "";

  //Prints topics that get modified.
  print "Writes to Topics:"
  inv @label . ({item} . pubAlter);
  print "";

  //Prints direct influences.
  print "Influenced By - Direct:";
  dirInf = direct . {item};
  inv @label . dirInf;
  print "";

  //Prints indirect influences.
  print "Influenced By - Indirect:";
  inInf = (indirect . (direct . {item})) - dirInf;
  inv @label . inInf;
  print "";
}
```

Figure C.6: Grok script that detects the publish alteration hotspot.

To detect this hotspot, lines 2, 3, and 4 get instances of direct and indirect communication between features. Line 2 gets instances of direct communication between a feature and a recipient feature's callback function. To do this, the publish and subscribe relations are joined by topic and then joined with the call relation to get the callback function that receives this data. Lastly, this is joined with contain relation on the left-hand side to get the parent class that sends the data. Line 3 does the same thing as line 2 except it lifts both ends of the relation up to the class level. Then line 4 gets the transitive closure of `indirect` to get all indirect instances of communication.

Now, callback functions need to be obtained. Line 7 gets all callback functions by joining the subscribe relation with call relation and then gets the range of that relation. Lines 8 and 9 takes a collection of relations and combines them together. This includes the `varWrite`, `varInfluence`, `varInfFunc`, `call`, and `write` relations. Then, the transitive closure of this relation is obtained. This gives a relation with the dataflow of all variables and functions in the program. This is stored in the `masterRel` relation.

Once the `masterRel` relation has been created, the publish alteration instances can be obtained. Line 12 does this by whittling down the masterRel relation to only start with callback functions and to only end with publishers. By doing this, it shows any callback functions that *eventually* result in a publication. The results are printed in line 13.

Because only printing the number of instances is not enough, lines 16 through 37 loop through each callback function in the set of publish alteration instances and print important details. In this loop, line 18 prints the name of the callback function being investigated. Then lines 22 and 23 print all the topics that get written to due to this callback function. This is done by taking the callback function and projecting it upon the `pubAlter` relation.

Once this is complete, the direct and indirect features that influence this callback function are printed. Lines 27 through 29 print the direct instances. This is done by taking the direct relations and projecting it on the callback function. This gives all the features that write to that callback function. Lines 33 through 35 print the indirect instances. This is done by getting all direct instances and projecting them on the indirect relation.