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 numerous teams, consist of millions of lines of code, and are split into numerous 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 modern vehicles are equipped with 100 million lines of code and are 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, automotive software needs to be inspected to verify that it is safe. The problem with this is that it can be difficult to carry out this detection; manual analysis does not scale well, search tools like `grep` have no contextual awareness of code, and code reviews, while effective, cannot target the entire codebase properly. Further, automotive systems are comprised of numerous, communicating features that can possible interact in unexpected or undefined ways. This thesis addresses this problem through the development of a static analysis methodology that detects custom problem areas coined as hotspots. We identify several classes of automotive hotspots that describe areas in automotive software that have the possibility of causing a feature interaction.

To detect these hotspots, this methodology employs a static, relational analysis toolchain that creates a queryable model from source code. Once these models are generated, relational algebra queries can be written that detect potential hotspots in the underlying source code. The purpose of this methodology is not to detect bugs with surety but rather scale down the amount of code that is inspected through other approaches.

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. Since 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 consisting of over a million lines of code (MLoC) divided across numerous components. To illustrate, Figure 1.1 highlights the sheer 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 Australian software testing firm Tricentis found that the global economic cost of faulty software was 1.1 trillion US dollars in 2016 [13]. While this amount currently compares to the gross domestic product of Mexico [14], the economic impact from such bugs is expected to increase as society continues to become even 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 calulated patient risk for medication perscription, miscalulated for 300,000 heart patients exposing them to the wrong 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].

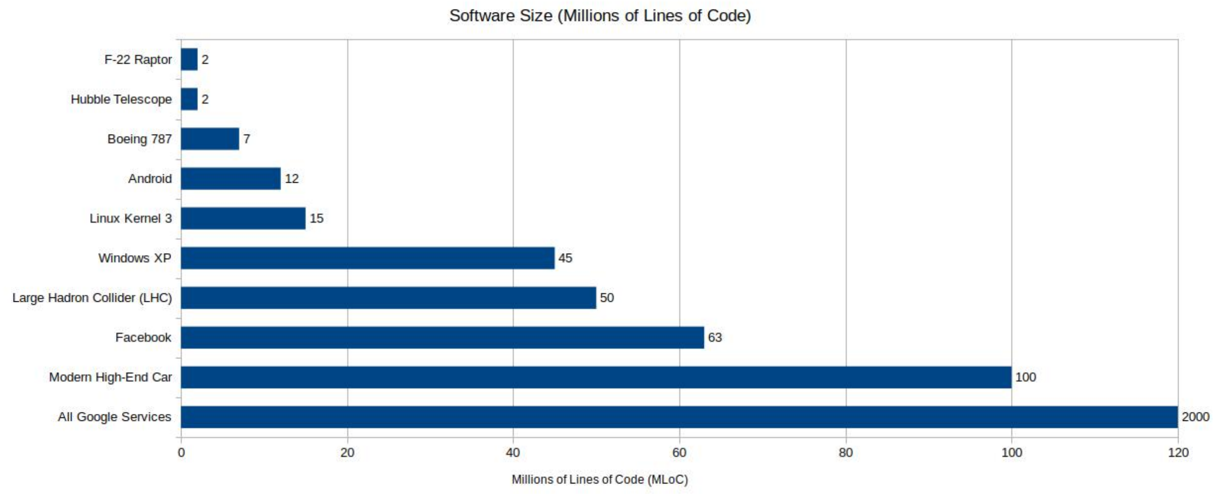


Figure 1.1: The number of lines of code for several software systems.

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]. While a lot of that 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 and ACC maintains a vehicle's speed based on a user set speed and the traffic around the vehicle. While these systems are designed to improve driver safety, there have been numerous cases of these features not operating properly. For instance, in 2017, the Fiat-Chrysler automotive group decided to recall over 1.25 million trucks due to a software glitch that could cause side airbags and seat pretensioners to fail during a collision [18]. These recalls have real consequences; the failure of these airbags have been linked to one death in the United States.

At a high level, automotive systems are comprised of numerous features that all communicate with each other. While the definition of a feature varies depending on the granularity at which someone decides to work at [19], I define a feature as "a coherent and identifiable bundle of system functionality that helps characterize the system 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 amongst two ADAS features in the blue car: collision avoidance (CA) and side collision avoidance (SCA). Here, one red car is merging onto the road causing the SCA feature in the blue car to move the car to the side to avoid a collision. The other red car 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. Due to the severity of vehicular feature interactions, I tackle the issue of detecting feature interactions in an automotive software system in this thesis.

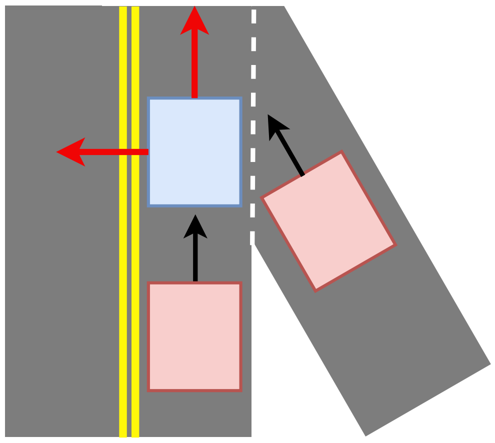


Figure 1.2: A case where vehicular features might interact in a vehicle (blue).

Detecting these feature interactions can be difficult due to codebase size and complexity. As this problem is not new, numerous techniques have been developed to assist developers. One popular technique is the use of manual code inspection to detect bugs and interactions. While code reviews are an important tool, they can be fairly slow. Effective code reviews should only focus on 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 [21]. Another detection method is the use of static analysis tools to detect problem areas and highlight calls between components [22]. 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. While valuable, these type of tools do not have the capability of detecting high-level patterns such as feature interactions.

The aim of this thesis is to present a novel static analysis toolchain to detect feature interaction hotspots in automotive software systems to avoid potential feature interactions. This novel 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. Since manual inspection is beneficial on a small scale, the goal of this approach is to scale down the number of areas required for manual analysis by identifying these program hotspots. I argue that this approach complements current tools used in the automotive domain due to its flexibility, scalability, and ability to detect message passing.

1.1 Hotspots

Before I give an overview of this thesis, the notion of a *hotspot* is introduced. A hotspot is an abstract concept that describes a potential problem area in code that can be detected using static analysis. The purpose of characterizing these hotspots is not to detect bugs with surety but rather to develop tools that can identify these areas so that afflicted code segments can be better inspected using other, more thorough, traditional analysis techniques. Since these problem areas differ depending on the domain, the type of hotspots one might want to look for varies greatly. For instance, a traditional software system may not have to deal with component communication or threads whereas an automotive system generally has to contend with lots of communication across ECUs. For this analysis methodology to be effective, users of this toolchain need to identify the hotspots they deem important prior to using the toolchain to analyze the software.

Figure 1.3 shows a potential hotspot for a monolithic software system. In this example, there are three separate components that each communicate with each other in some way. Component 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 was detected by the toolchain, system experts could then verify that the interaction between these three functions operates as expected.

Static analyses are evaluated with respect to their precison and recall of the target information [23][22]. Developing tools that identify hotspots in a software project are no exception; different tools may identify problem areas with varying levels of precision and recall. 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 [24] and developing a "golden" static analysis tool that identifies hotspots with high precision and

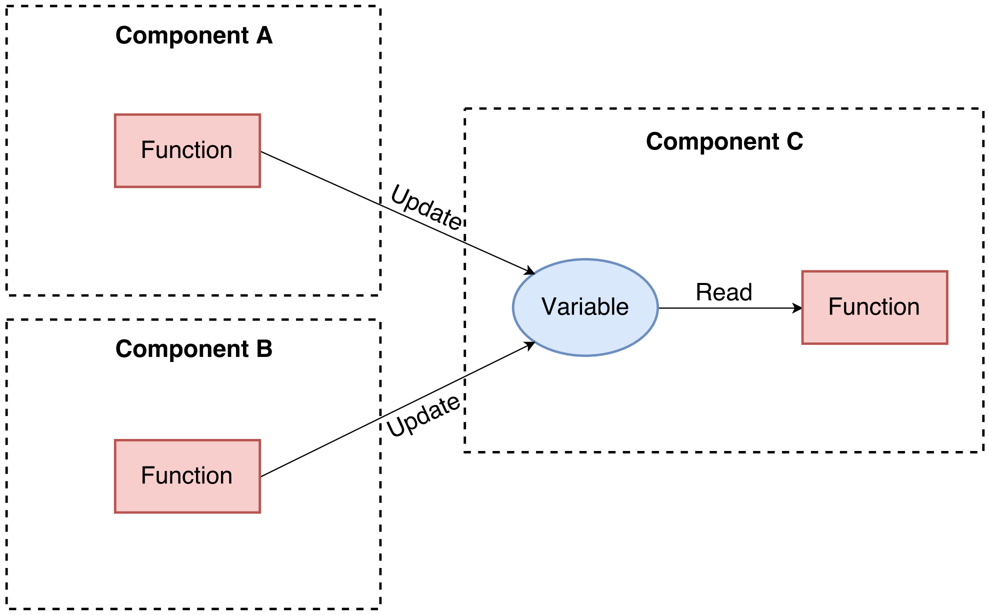


Figure 1.3: An example of a possible hotspot in a monolithic system.

recall is impossible. As such, it is best to identify hotspots that have high recall since any false positives returned by some static analysis tool can be filtered out using other static analysis methods or manual analysis. While it may seem tedious to pair this methodology with other static analysis tools, research by Willis et al. [25] 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 extremely 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.

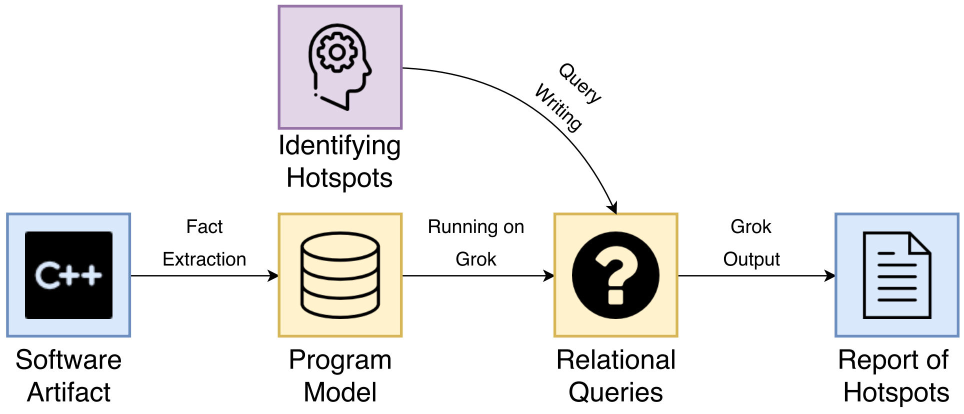


Figure 1.4: The hotspot detection toolchain.

1.2.1 Generation of Program Model

A high-level overview of my 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 since 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. Since software projects vary in programming language, different fact extractors must exist for each language. While 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, two custom fact extractors called ClangEx and Rex 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 distributed robotic systems.

1.2.2 Generation of Hotspot Report

Once a model exists that describes the automotive software, 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, I 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 comb through the program model looking for these hotspots. These queries are written in a specialized query language known as Grok (see Section 2.2.3) and output a list of areas of code that reflect the hotspot description. I develop Grok scripts that detect each hotspot described in this thesis and run 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. While these hotspot reports do not necessarily show areas of code that contain problems, these reports can be used in tandem with manual code reviews or additional static 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, I 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. The precision and recall of the detection of these hotspots is explored.

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 the unique challenges associated with using static analysis on software in the automotive domain. Further, it describes several different types of hotspots that might be present in such systems.

Chapter 5 describes a major case study that was conducted that demonstrates the feasibility of detecting hotspots using the relational analysis toochain 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.

In addition to the fact extractors discussed in Chapter 3, Appendix A is a user manual for the bfx64 extractor and Appendix B is a user manual for the ClangEx and Rex extractors. 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 my thesis to detect hotspots.

2.1 Related Work & Tools

Static analysis is not a new concept. Research into static analysis techniques spans decades and has become an essential part of the software development lifecycle. 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 is continuing, 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 two relevant static analysis tools that are used in industry. Section 2.1.1 looks at research that developed unique approaches to statically detect interactions between components in software artifacts. 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 novel approaches to efficiently analyze software systems. This section explores two state-of-the-art techniques to statically detect interactions amongst components in distributed environments; **Jianta** and a method to analyze **distributed robotic systems**. Other areas of related work are briefly discussed.

Jianta

Tsutano et al. developed a novel analysis framework that targets the applications built for the Android platform to detect interactions amongst themselves [1]. Known as Jianta, this framework works 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 from the Java Class Loader and analysis engines that interface with the analysis controller. Once information is obtained from the ADB, Jianta is able to generate graphs showing component communication. Figure 2.1 shows a high-level overview of this architecture.

Compared to previous research, this approach is powerful since previous solutions to analyze cross-application communications required combining the Android packages (APKs) into a single APK and then extracting inter-component communication (ICC) calls. Instead, by leveraging the class loader, Jianta is able to generate class, method graphs, 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.

While Tsutano et al. developed an innovative way to detect application interactions on the Android platform, techniques to analyze these graphs is out of scope for their research. Notably, this technique focuses on extraction and visualization of Android application information using external tools such as GraphViz [26] or TraViz [27]. While it would be feasible to analyze or visualize smaller projects, determining problematic interactions between applications in large projects may not be manageable without having access to some query mechanism.

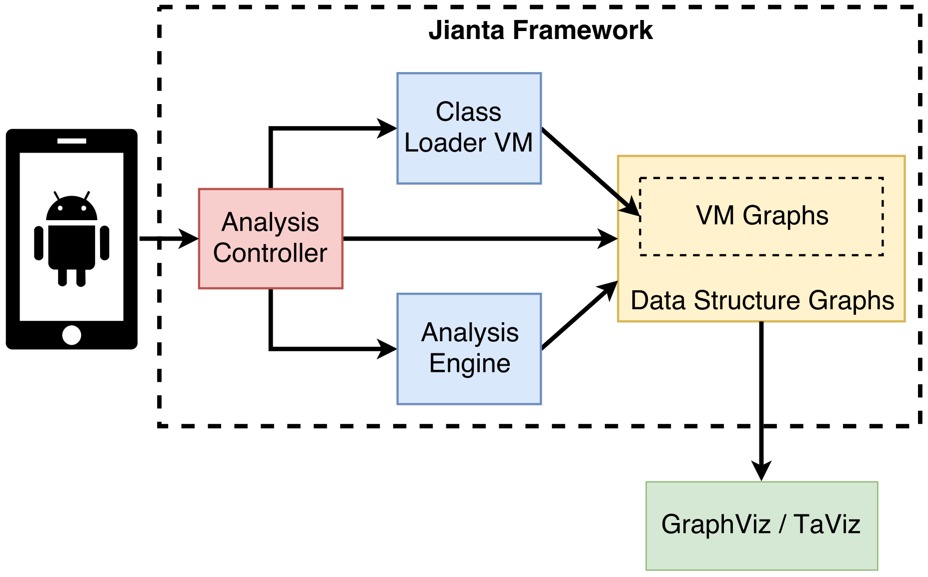


Figure 2.1: The Jianta analysis architecture. Adapted from [1].

Distributed Robotic Analysis

Purandare et al. developed an efficient method to extract conditional component dependence from robotic systems [2]. Analysis using their approach looks at interactions between components that arise due to program control flow decisions. This technique discovers interactions that are created due to control structures or interactions that are the effect of other interactions. The result of this analysis is a single model that highlights any interactions that arise in some project or 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 the interactions. Figure 2.2 shows the architecture of this static analysis technique.

Similar to Tsutano et al., Purandare et al.'s research focuses on extracting information from software artifacts. Therefore, in large projects, it can be difficult to sift through models generated by this approach to find specific instances where components interact with each other. Unlike Jianta, a portion of this framework is devoted to comparinchg multiple models to detect how code changes affect interactions between components. 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]..

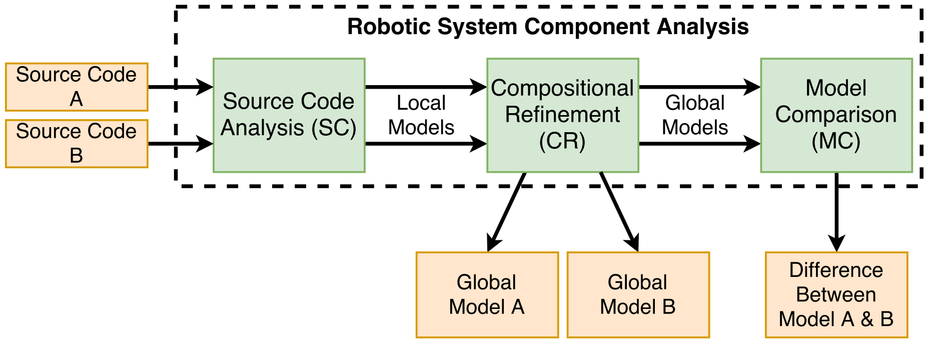

Robotic System Component Analysis
Source Code A
Source Code B
Source Code Analysis (SC)
Local Models
Compositional Refinement (CR)
Global Models
Model Comparison (MC)
Global Model A
Global Model B
Difference Between Model A & B

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

Other Work

In addition to work by Tsutano et al, and Purandare et al., there is other research in the static analysis field that aims to detect interactions between subsystems in event-based systems. For the purpose of this section, event-based systems are systems which produce, consume, and react to events send between components [28]. Safi et al. [29] develop an approach for detecting anomalies in event-based systems. Since these systems tend to suffer from unexpected interactions that arise due to nondeterminism, Safi et al. present a static analysis technique called DEvA to detect potential event anomalies. DEvA was tested on 20 open-source systems and found event-based anomalies in each.

Jayaram et al. [30] develop program analysis techniques for event-based distributed systems. In their research, they describe three different types of static analysis checks that can be integrated into languages that utilize event-driven semantics; immutability analysis, guard analysis, and causality analysis. Each type of analysis can be used in different types of event-based systems including those that use the publisher/subscriber architecture. Jayaram et al. found that integrating such checks into a programming language reduced the number of manual checks programmers had to do when writing event-driven programs.

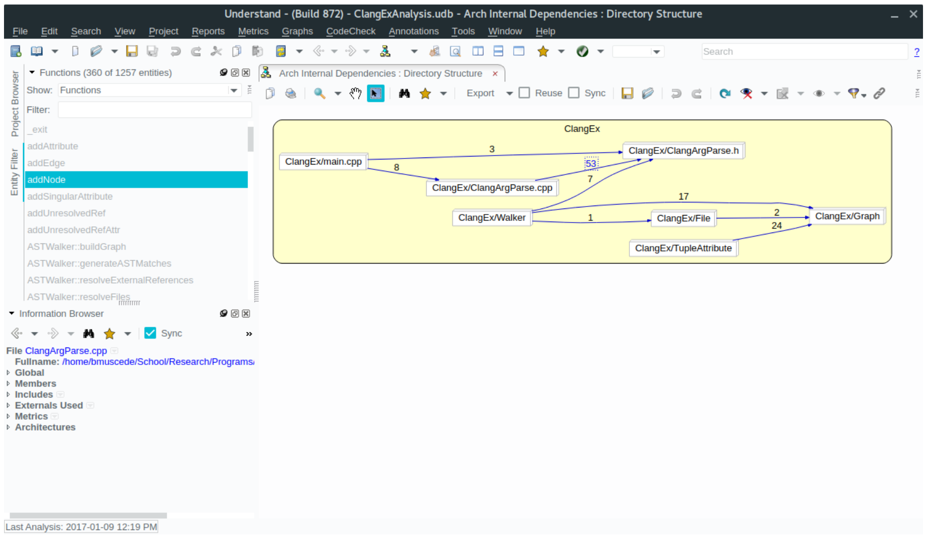
Other researchers have attempted to use forms of relational algebra in defect detection and static analysis. Kozen [31] proposes using a form of relational algebra called Kleene Algebra with Tests (KAT) to statically verify the compliance of safety policies. While Kozen demonstrates that KAT is effective in verifying such policies, programs analyzed using this method need to be translated to an equivalent automation.

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 commerical static code analysis integrated development environment (IDE) created by Scientific Toolworks. The tool features incorporated code metric reporting, simple error detection, and can generate UML class and dependency diagrams of the underlying project [32]. 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 that they are defined in.



Understand - (Build 872) - ClangExAnalysis.udb - Arch Internal Dependencies : Directory Structure
File Edit Search View Project Reports Metrics Graphs CodeCheck Annotations Tools Window Help
Project Browser
Entity Filter
Arch Internal Dependencies : Directory Structure
Export Reuse Sync
Information Browser
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

Functions (360 of 1257 entities)
Show: Functions
Filter:

_exit
addAttribute
addEdge
addNode
addSingularAttribute
addUnresolvedRef
addUnresolvedRefAttr
ASTWalker::buildGraph
ASTWalker::generateASTMatches
ASTWalker::resolveExternalReferences
ASTWalker::resolveFiles

File ClangArgParse.cpp
Fullname: /home/bmuscede/School/Research/Programs/
Global
Members
Includes
Externals Used
Metrics
Architectures

Last Analysis: 2017-01-09 12:19 PM

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

There are two major features in Understand that make it a valuable tool. First, it has a feature called *CodeCheck* which 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, there is a limitation to the types of issues that CodeCheck can detect as it has trouble performing dataflow analysis, memory allocation/leak detection, and tracking variable use amongst functions [33]. The other major feature is the 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. An issue with this is that, in large software projects, these UML diagrams can become restrictively large and incomprehensible.

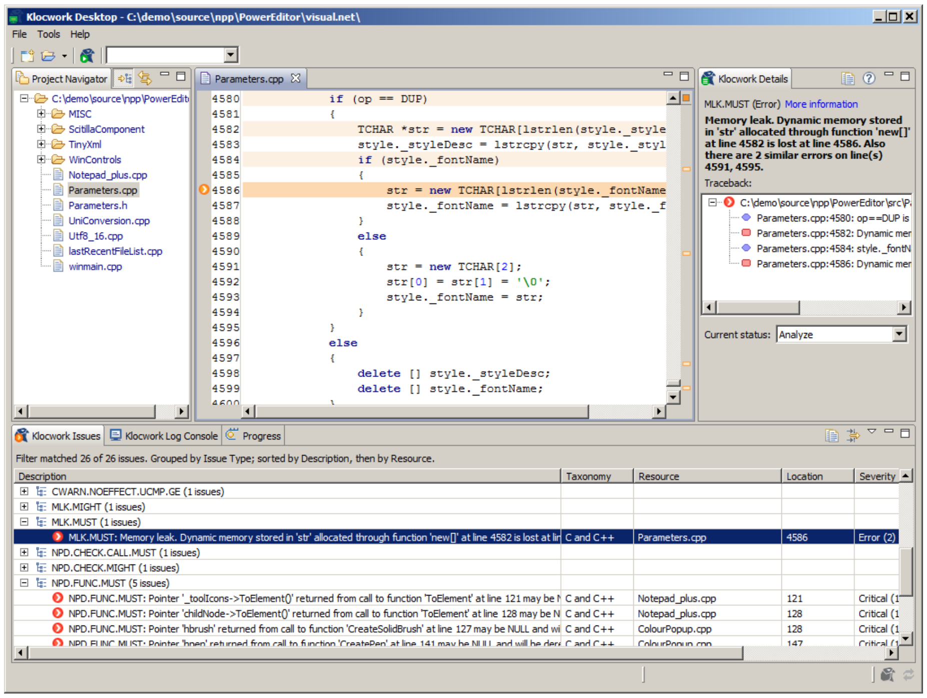
Klocwork

Klocwork is a commercial 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 [34]. 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 four causing problems. Further, both Klocwork tools allow for users to develop their own fingerprints. These custom fingerprints target the 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.

While 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 means that users cannot easily write fingerprints that check how specific programming frameworks (such as Hadoop [35] or Boost [36]) are used. Further, while Klocwork has 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 since 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 comprised of several steps that generate a queryable model that represent some underlying software artifact. The toolchain is used in this thesis to identify hotspots in an automotive software system. Figure 2.5 shows each step in the



Klocwork Desktop - C:\demo\source\npp\PowerEditor\visual.net\

File Tools Help

Project Navigator

- C:\demo\source\npp\PowerEdit
  - MISC
  - ScitillaComponent
  - TinyXml
  - WinControls
  - Notepad_plus.cpp
  - Parameters.cpp
  - Parameters.h
  - UniConversion.cpp
  - Utf8_16.cpp
  - lastRecentFileList.cpp
  - winmain.cpp

Parameters.cpp

```
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;
```

Klocwork Details

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

Klocwork Issues | Klocwork Log Console | Progress

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].

relational analysis toolchain. Gray boxes represent files, blue boxes represent programs, and the single green box represents additional programs that could be amended to the toolchain that are able to read and write factbases generated through the "fact extraction" step.

The basic idea of this toolchain is to generate a queryable model that represents some software artifact. These models are created automatically by feeding software artifact into a custom, human-defined analyzer called a **fact extractor**. The models generated by fact extractors are a collection of facts about the software system that come together to form a factbase. Facts can be any desired information about the software artifact; functions, the relationship between a class and a functions, or variables all could be considered facts. To represent these models, the toolchain uses a common, plain-text format known as tuple-attribute (TA). When developing a fact extractor, the user should have it only extract facts of interest; including too information means that the model is too detailed for lightweight analysis. For instance, if a user is only interested in examining the function calls of a program, the queryable model of that program's source code should just include 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 generic. It can be adapted to work with any software artifact and can be extended so that tools other than Grok and LSEdit can read and write to these models. As an example of this genericity, Java [37], C++ [38] and ELF object files [39] all can currently be analyzed using this toolchain. In addition, with the TA encoding being simple to read and write to, tools can be added that load in TA factbases and output modified models or supplemental information.

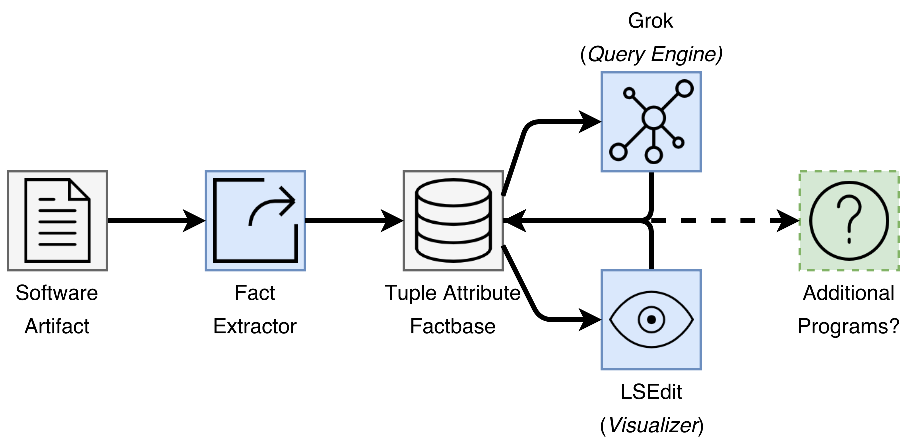


Figure 2.5: The relational algebra toolchain.

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. Since 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 only extracts the function call graph of a program whereas another C++ extractor that might exist to extract all static variables.

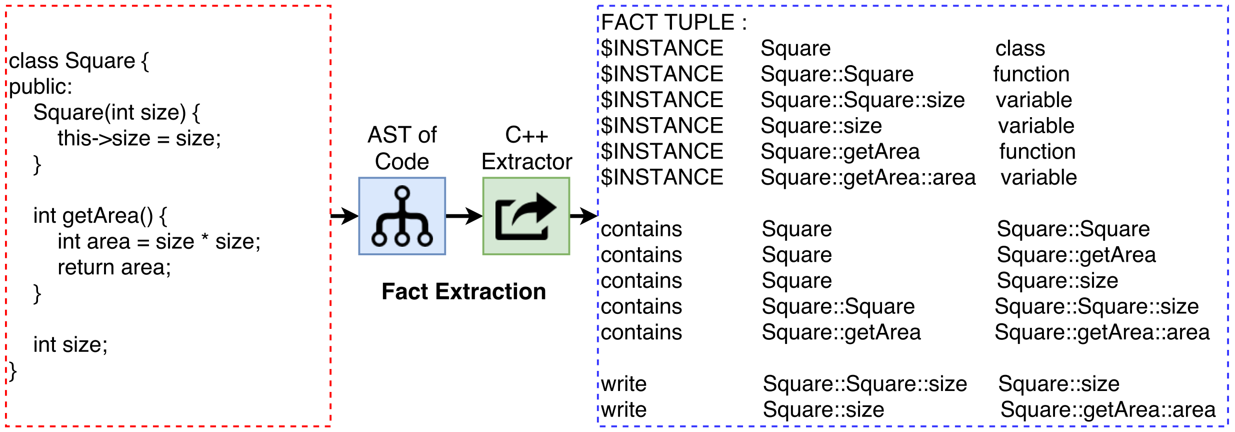
While it may appear daunting to develop a fact extractor to target different software artifacts and different query types, the hardest part of developing these extractors is determining the *type* of information one wants to extract and how that information should be best exploited. Once this information 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, the abstract syntax tree of parsed source code, or even information from XML log files. 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.

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 program consists of a simple class called `Square` that represents square objects (shown on left of Figure 2.6). This class can calculate the area of the square through a function called `getArea`. When this code snippet is converted into an abstract syntax tree and then processed by this example fact extractor, a resultant TA model is generated (shown on the right). This 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 next section describes the layout of these TA files and how data is presented.

2.2.2 Factbases & The Tuple-Attribute Language

Once extracted, facts are stored in factbases in a format known as tuple-attribute (TA). First developed by Dr. Ric Holt in 1997, the TA language is a formal grammar designed to describe directed graphs (digraphs) in a simplistic yet powerful format [40] because one mental model that developers have of software systems tend to be a graph consisting of a collection of sub-components connected by edges denoting their relationships. Graphs encoded in the TA language can then be transformed through a series of relational operators as defined by Tarski's relational algebra [41]. These transformations produce new facts that can answer questions about the underlying software system.

Factbases encoded in the TA language are written in plain text and are divided into two



```
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.

major sections; tuples and attributes. Tuples describe entities and relationships amongst 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. Table 2.1 shows each section of a TA file along with its purpose.

	Section Name	**Section Header**	**Description**
Scheme	Tuple Schema	SCHEME TUPLE	Describes the schema for tuples.
	Attribute Schema	SCHEME ATTRIBUTE	Describes the schema for attributes.
Fact	Tuple Facts	FACT TUPLE	This section stores facts that describe the software artifact.
	Attribute Facts	FACT ATTRIBUTE	This section stores attributes for already-defined facts.

Table 2.1: The different sections of a TA file.

In a TA file, each of these four sections need to be declared using a special keyword that is written before any element of that section is written. 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 tuples 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.

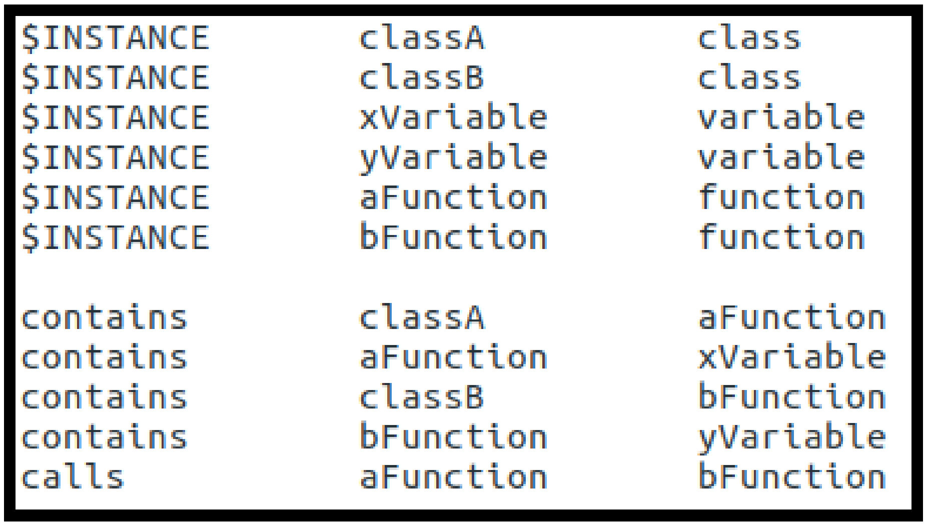
Tuples

The tuple section of a TA factbase is where entities and the relationships amongst entities are defined. Tuples follow the Rigi Standard Format (RSF) encoding where they are triplets [42] of the form: `Item1 Item2 Item3`. Both entities and relationships are written in this RSF format.

Entities are encoded in 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. Last, `<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. They are written in the form: `<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 participate 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.

Attributes

The attribute section describes attributes for previously defined entities and relations in the TA model. Attributes are untyped key-value pairs which can either be unique for a single entry or applied to the entire entity or relationship set [43]. There are two types of attributes: single-value and multiple-value attributes. Attributes are important for storing information that might not fit as an entity or relation. An example of this could be a boolean flag that notes whether a variable is static or not.

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 $key = (value1\ value2\ ...)$.

Figure 2.9 shows several example lines of the attribute section of the TA file. In it, there are several entities and one relationship. With `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 that it is used in. The calls relation between `aFunction` and `bFunction` has an attribute called numberOfCalls.

```
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.

2.2.3 Relational Query Engine - Grok

Grok is a relational algebra calculator that allows users to execute relational algebra queries on TA factbases. 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 new facts to the factbase [43]. Relational operations that can be performed include transitive closure, relational selection, and composition[1]. 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 [43] to allow users to write queries that would allow for relation lifting and edge induction to improve the comprehenison of large software systems. Since then, the use of Grok has expanded to other things such as clone detecton in source code [44].

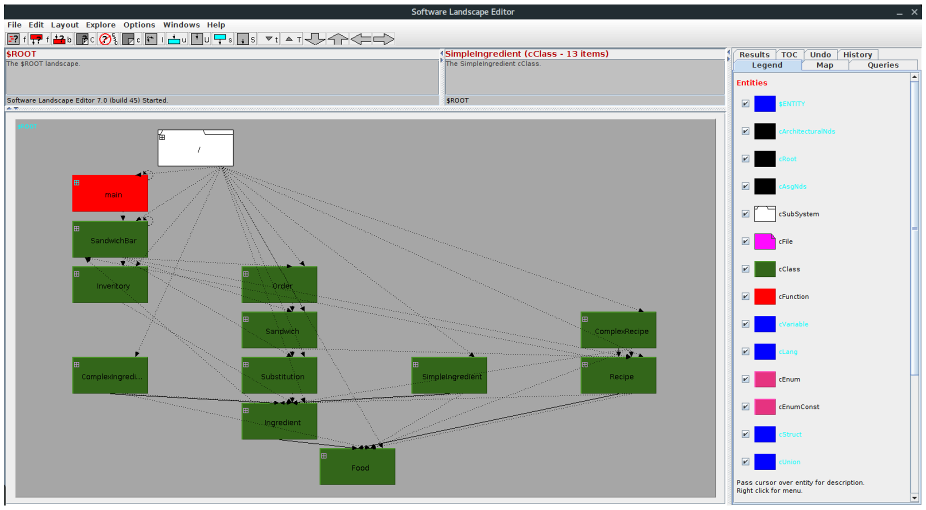
In addition to Grok, a Java-based version called jGrok was developed in 2006 [45]. While jGrok has a similar syntax to Grok and supports the same operations, there are some subtle differences. The benefit of jGrok is that it supports several additional operations such as better math and file operations. However, one drawback is that jGrok is slower compared to Grok due to fewer performance optimizations and because it runs in the JVM.

2.2.4 Model Visualizer - LSEdit

The Landscape Editor (LSEdit) is a tool that operates on TA factbases to visualize, manipulate, and clustering of digraphs [46]. 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

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

coloured boxes and relationships are shown as lines between the boxes. In this particular visualization, relationships amongst C++ classes (green) in the ClangEx fact extractor project are shown.



$ROOT
The $ROOT landscape.
Software Landscape Editor 7.0 (build 45) Started.

SimpleIngredient (cClass - 13 items)
The SimpleIngredient cClass.
$ROOT

Pass cursor over entity for description.
Right click for menu.

Figure 2.10: LSEdit visualizing the ClangEx fact extractor project.

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 numerous autonomous robotic applications including the Autonomoose self-driving platform. It is important to understand how ROS works since 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, interprocess communication, and package management [47]. 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 [48].

The remainder of this section will describe the communication framework of ROS. While ROS has other services, Autonomoose makes heavy use of the communication framework. Understanding ROS' communication framework is important to understand 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 which 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.

While Figure 2.11 shows topics as an intermediate buffer that exists outside of each component, topics are simply named buses in which messages flow [47]. ROS simply 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 passes 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 is 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

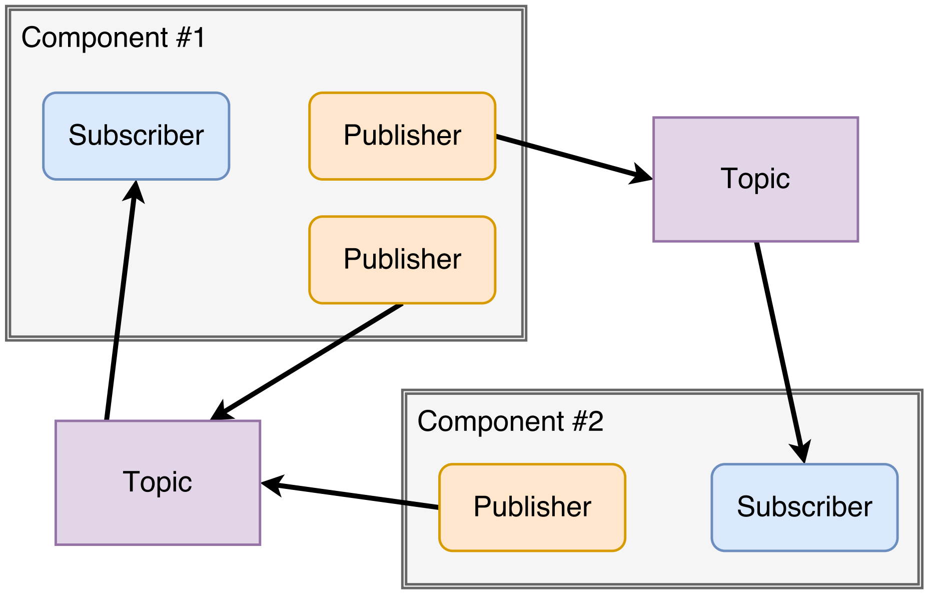


Figure 2.11: An example of the publisher-subscriber paradigm in ROS with two components.

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.

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 representing that project needs to be created by running the source code through a fact extractor. For the toolchain to be effective, fact extractors need to be developed that can generate high quality, accurate models that contain just the right amount of information that allow for the detection of hotspots. Fact extractors operate by automatically discerning information about a software artifact based on a predefined schema that describes the type of entities, relationships, and attributes it extracts.

For this thesis, I 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. While 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 effective for different types of situations.

While I developed ClangEx and Rex, numerous other extractors have been developed through previous research. Further, since 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 [49]. Three notable extractors developed prior to this thesis are **bfx64**, **ASX** and **CPPX**; 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 [38]. 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. While I 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] [50]. By reverse engineering debugging information, ASX is able to gather information about functions, variables, classes, and item usage. While ASX is useful for assisting developers with program comphrehension, it has also been used in other research to detect and eliminate dead functions in a software project [51]. ASX was important in developing ClangEx as I used it as a "sanity" check to compare results between it and ClangEx. ASX and ClangEx models of example code were compared ensure output was comparable[2]. 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 I developed this extractor while conducting my 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 [45] and targets ELF object files. While bfx64 is capable of targeting C and C++ code, the models generated were not precise enough for feature interaction hotspot detection. Since 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 digraph to TA conversion algorithm.

Table 3.1 compares the bfx64, ClangEx, and Rex extractor. 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, the *Purpose* column indicates the specific purpose of each extractor.

The remainder of this chapter introduces the ClangEx extractor in Section 3.1 and the Rex extractor in Section 3.2. For each extractor I present the motivation behind its devel-

[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.

[2]Differences in entity naming did exist since ASX and ClangEx have different naming schemes.

Comparison of Fact Extractors			
Name	*Language(s)*	*Compiltation Level*	*Purpose*
bfx64	C, C++	ELF Object Files	For analyzing projects where source code is not available. This includes third-party supplier code that is non-obfuscated.
ClangEx	C, C++	Abstract Syntax Tree	General C and C++ fact extractor that can be extended. Captures basic C and C++ language information in source code passed to it.
Rex	C, C++	Abstract Syntax Tree	For analyzing ROS-based projects to detect messages sent between components. Also captures flow of data between functions and components.

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

opment, its tuple-attribute schema, and its internals. Further, each section discusses the benefits and drawbacks of using the extractor. While not a part of this chapter, Appendix A provides installation and usage instructions for the bfx64 extractor and Appendix B provides installation and usage instructions for the ClangEx and Rex extractors.

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. Unlike bfx64, which targets object files produced from compilation, ClangEx analyzes abstract syntax tree (AST) fragments created during the compilation of C and C++ source code. The reason ClangEx uses the Clang compiler for analysis is that Clang has 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 remaining generic, ClangEx serves as a model extractor that can be 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.

Due to the idiosyncrasies of processing C and C++, ClangEx has two main analysis modes: (i) regular mode; and (ii) full-analysis mode. The purpose of these two modes is to utilize two different source code processing methodologies. While both modes use the schema shown in Figure 3.1, models generated can be drastically different from one another due to how these modes deal with header files. Regular mode examines all features inside a C or C++ source file and ignores header files. This ensures that any unnecessary declarations contained in header files not directly related to the source file are not included in the model. Opposite to this, full-analysis mode 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 "blob-like" model since 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 used depends on the types of queries one wants answered and the number of files being processed. For instance, if a user is analyzing the whole project, full-analysis mode is recommended since the extractor will encounter all definitions for all header file declarations. Further, if a function is declared and defined inside a header file, 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.

How the extractor records an instance of the language feature depends on the type of language feature being recorded. These instances can be recorded as nodes, relationships, or attributes in the resultant 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 while 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. Each `union` and `struct` has an attribute called *isAnonymous* that indicates whether the declaration is anonymous or not. Overall, most nodes pertaining to some language features will have several attributes that describe that instance. 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 that each of these language node types partake in. 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`. Overall, this relationship is a "catch-all" for 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 representing function calls. This relationship forms the call graph for the project that the model represents.

In addition to nodes that describe instances of C or C++ language features, models

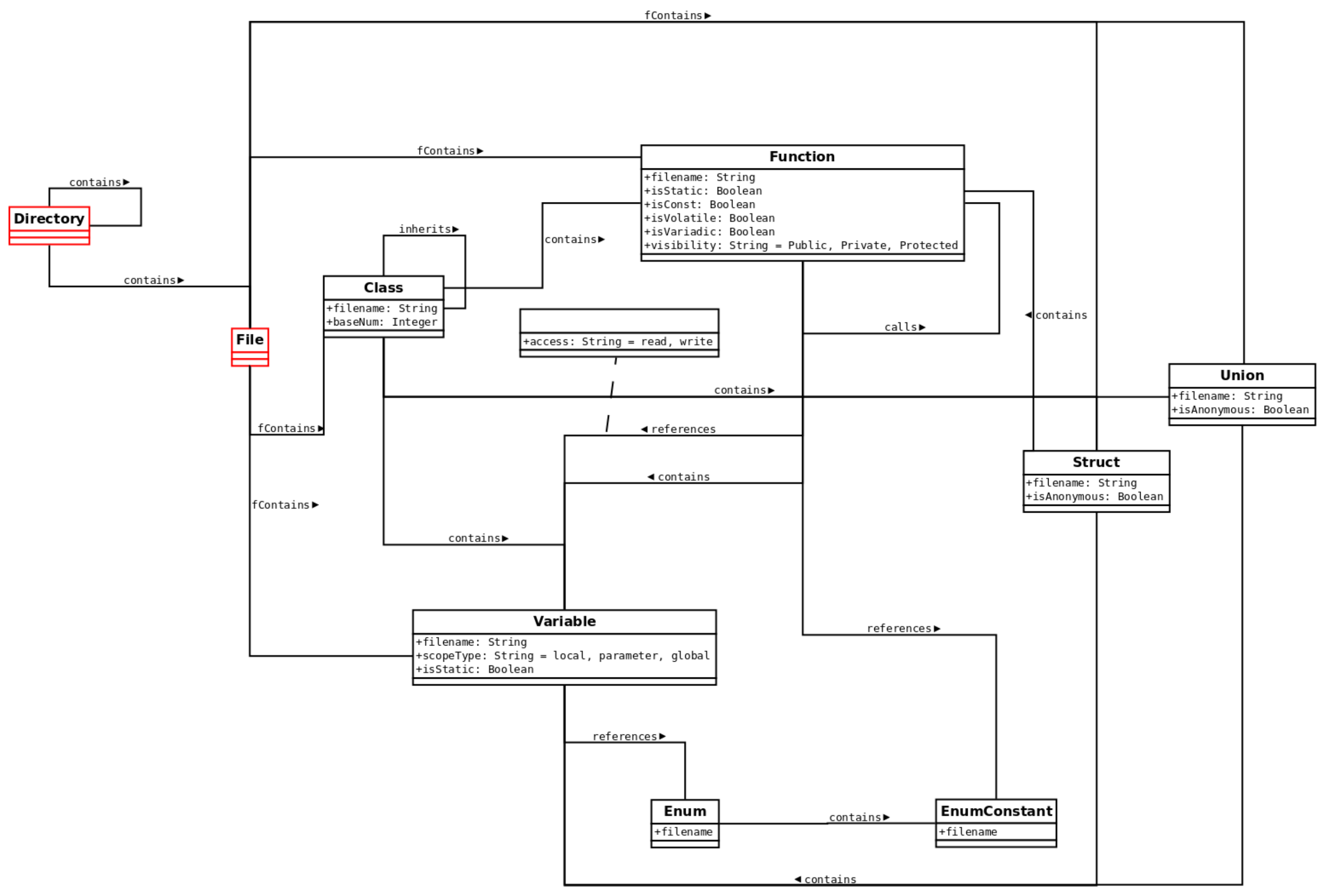


Figure 3.1: The ClangEx metamodel. Black classes are C/C++ language features.

produced using ClangEx also contain nodes that describe files and directories. Essentially, a directory node is a single TA fact that represents a single directory in a file system and can possibly be contained inside other directories (via the `contain`) relation. File nodes represent files in a file system and can be contained inside directories. File nodes are facts that either represent source or header files. This structure mirrors the file/directory structure maintained by the operating system. ClangEx does not use the `contains` relation to link files and C/C++ language features together. Rather, for each C and C++ language feature node (`struct`, `class`, etc.), there is one or more special `fContain` (file contains) relations that point from some file node to that C/C++ node. The reason the `contains` relation cannot be used is because a node can only be "contained" inside one container at a time. Since C or C++ language features may be declared in one file and defined inside another, the `fContain` relation is used instead.

3.1.2 Advantages of the ClangEx Extractor

While other C and C++ extractors exist, ClangEx is unique for several reasons. First, by having access to the Clang API, ClangEx is able to obtain detailed information about source code as parsed. This gives ClangEx the ability to generate detailed models of source code easily. Further, because Clang has an open-source API, additional detection functionality can be integrated into ClangEx easily.

Since ClangEx obtains information from the AST of a source file, ClangEx does not require a project to be linkable or even fully compilable. In projects with errors, 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. An advantage of not requiring the project to be linkable is that if libraries a project requires are not present, ClangEx can still generate a detailed model of that project. The extractor performs its own manual linking process to connect 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. As a result, ClangEx simply deletes any relationship that uses that reference from the graph.

3.1.3 Disadvantages of the ClangEx Extractor

ClangEx also has several disadvantages and situations where it is not useful. A major disadvantage is that ClangEx is bound to the Clang compiler. While this may not appear to be an issue, Clang is not able to compile all C/C++ programs out of the box. While

Clang is adherent to ISO standard C and C++ and supports up to C11 and C++14 [52], it 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` [53]. The implications of this is that 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. One such gcc-ism used heavily in the Linux 3 kernel that Clang cannot process are `asm goto` statements.

Second, ClangEx does not incorporate into existing build workflows very well. To pass compiler arguments to Clang, ClangEx utilizes a Clang compilation database to determine the compiler flags for each file. These databases are JSON databases that contain each source file for a project as well as all compilation flags required to build each file. The reason for the use of these databases in ClangEx is because ClangEx is not a standalone compiler and cannot be used as a replacement to Clang. Instead, ClangEx has its own frontend that allows it to interface with Clang. 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[3]) to generate these databases. As such, it may take some effort to analyze a large-scale project with ClangEx since developers will need to compile their whole project once 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. As a result, ClangEx does not parse or traverse the AST; this is handled by Clang. Only when Clang encounters a C or C++ language feature that ClangEx finds desirable does Clang pass over control of the AST to ClangEx. To do this, Clang invokes a specific ClangEx function and passes over an AST node object that represents the AST node ClangEx finds desirable.

ClangEx is divided into three major sections that are responsible for generating a TA model from source code. These are: (i) the Driver module that carries out command-line operations and activates Clang; (ii) AST Walker modules that carry out logic to process desired sections of the AST; and (iii) 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. From a high-level perspective, the Driver module takes commands from the user and starts the user’s desired AST walker. The

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

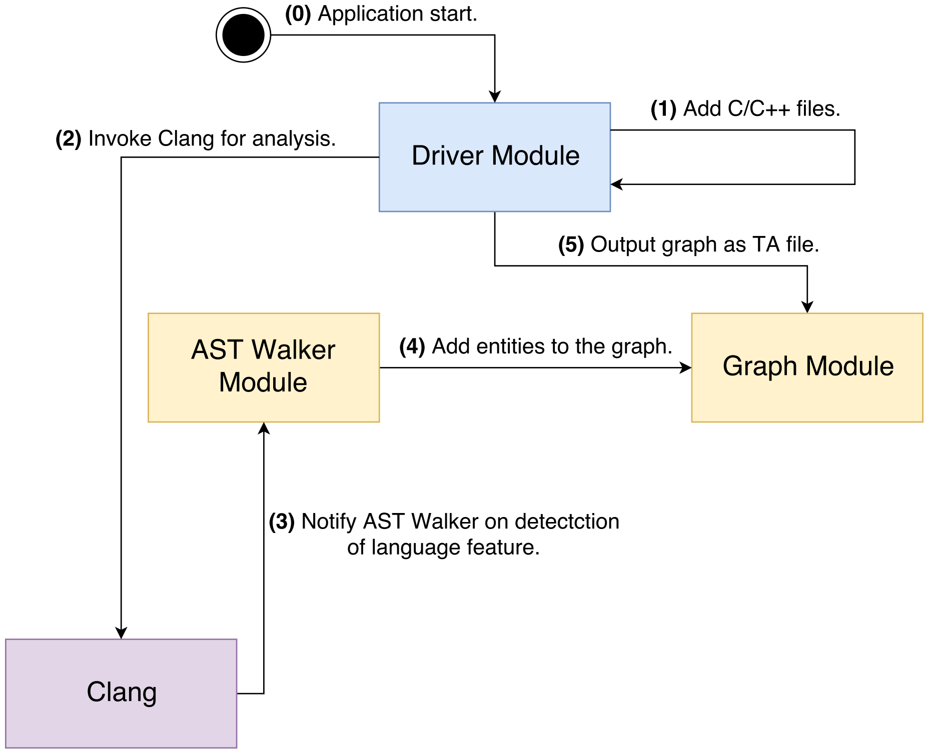

(0) Application start.
Driver Module
(1) Add C/C++ files.
(2) Invoke Clang for analysis.
(5) Output graph as TA file.
AST Walker Module
(4) Add entities to the graph.
Graph Module
(3) Notify AST Walker on detectction of language feature.
Clang

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

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. Once complete, the Driver sets up the selected Walker 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 an in-memory digraph through the Graph module. Once Clang finishes parsing all specified source files, ClangEx attempts to resolve any undefined or unknown references contained in the in-memory digraph. This resolution process happens only when all source files have been parsed. If a reference still cannot be resolved after this step, that reference is deleted from the pool of undefined references and not included in the final digraph. The reason a pool of unresolved references is retained is because if one source file is analyzed before another, any references to items in source files not yet processed will not yet exist in the graph. Since the source files can be processed in any order, references added to this pool are retained until the final reference resolution phase.

ClangEx does not have much control in the way the source file is analyzed. Rather, most of the time when ClangEx is operating, it is running Clang-specific code that generates and processes the AST. Only when Clang finds a “match” on a particular portion of the AST does the AST Walker module process that item and add it to the graph.

The remainder of this section will discuss both AST Walker modules and the Graph module in detail. The Driver module will not be covered here as it simply takes in command line arguments and invokes Clang. For commands and how to use ClangEx, see Appendix B.

AST Walker Modules

An AST Walker is a component which “walks” through the AST as it is being processed and runs specific code when specific language elements are encountered. While AST Walkers vary in behaviour, all AST Walkers have two specific elements; a section where AST matchers are defined and a function called `run` that is invoked whenever a specific language element is encountered. AST matchers are special statements that are defined prior to the AST Walker being run that tell Clang which portions of the AST that ClangEx is interested in. AST matchers are used in ClangEx because it allows Walkers to be able to toggle on and off different language features at a user’s request. If a user does not want to extract details about a certain feature (such as `enums` or `classes`), ClangEx will not activate that matcher. If that happens, Clang will not call ClangEx even if that specific AST fragment is encountered.

In the case of ClangEx, there are two different AST Walkers a user can select that each correspond to a different type of analysis mode; regular and full-analysis. Each of these two Walkers have their own set of AST matchers and their own specialized code which runs when a match is found. 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.

AST Matchers		
	Full-Analysis Mode	*Regular Mode*
Classes	Matches any class declaration statement. `class A` would be detected.	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.

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[4].

[4]System header files are not processed since processing all code fragments inside all system headers would generate models that contain far too much irrelevant information.

The reason ClangEx has these two processing modes is due to how header and source files are used in C and C++. When compiling a project, a user specifies the *source* files they want compiled. From there, the preprocessor includes all header file code included at the top of the source file before generating the AST. Most of these included header files either come from system files or declare code that is defined in other source files. This information does not need to be processed by ClangEx for the current source file since it does not relate. The problem here is that ClangEx has no way of knowing which header files describe elements inside the current source file. Even if ClangEx tried to overcome this issue by only processing the header file with the same name as the current source file, there is the possibility that a user broke convention and named the header file differently. As such, either all header files need to be processed (full-analysis mode) or none do (regular mode).

Figure 3.3 shows a portion of the AST matcher code from the full-analysis AST Walker which tells Clang which AST fragments to match on. This figure is showing several AST matchers that match on `functions` and `classes`. These matchers detect when portions of the AST match **exactly** to the statement described by the matcher. As an example, take line 8 in Figure 3.3. This matcher detects a C or C++ call expressions: it binds a variable to be the call-expression AST object and another variable to be the AST function object that indicates the function where the call originated.

When Clang detects that an AST matcher has been triggered, it invokes AST Walker code that adds associated nodes or edges to the digraph contained in the Graph module. While this may appear to be a straightforward task, there are several things that the Walker module has to do prior to adding a graph item. First, it has to generate a unique and linkable ID for the item. This ID has to be the same regardless of where this item was encountered so that it can be linked across compilation units. For edges, the ID simply consists of the source node ID combined with the destination node ID. While developing a unique node ID is not as difficult in C, this 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 it 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 gets the next ancestor AST node that is a declaration. It performs the same steps to get the ID of that declaration and then prepends it to the front of the initial ID that ClangEx is trying to generate. 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 for both languages in this manner. Figure 3.4 shows how an ID is generated for a local variable inside a function. Here, ClangEx generating an ID for the `testVar` local variable. Although local variables

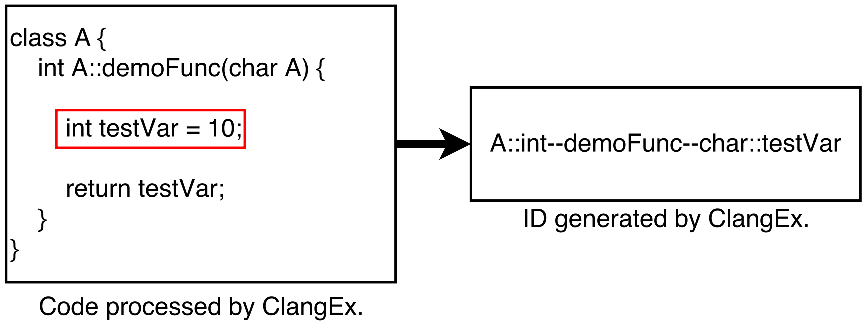
```
void BlobWalker::generateASTMatches(MatchFinder *finder) {
    //Function methods.
    if (!exclusions.cFunction) {
        //Finds function declarations for current C/C++ file.
        finder->addMatcher(functionDecl(isDefinition()).bind(types[FUNC_DEC
    ]), this);

        //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.

do not have qualified names, this keeps the testVar node ID unique.

In addition to generating an ID, the Walker must generate attributes for an 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. This filenames attribute is appended to each time the same AST node is encountered inside a new header or source file. Once all of this information is collected, ClangEx adds the node or edge to the graph using 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 adds 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



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

      int testVar = 10;

      return testVar;
   }
}
```

```
A::int--demoFunc--char::testVar
```

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

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 exist. Here, two edge list entries could be resolved and one could not because the `functionB` function does not exist in the digraph.

Graph Module

The Graph module maintains a digraph of all encountered nodes and edges while ClangEx is running. This digraph uses **ClangNodes**, each of which describe some C or C++ language feature, a directory, or a file; and uses **ClangEdges**, each of which represents a relationship between ClangNodes. Both ClangNodes and ClangEdges have an ID, label, and type that describe them.

The Graph module 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. While 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 also is responsible for converting the digraph into a TA model. Algorithm 1 shows the algorithm responsible for

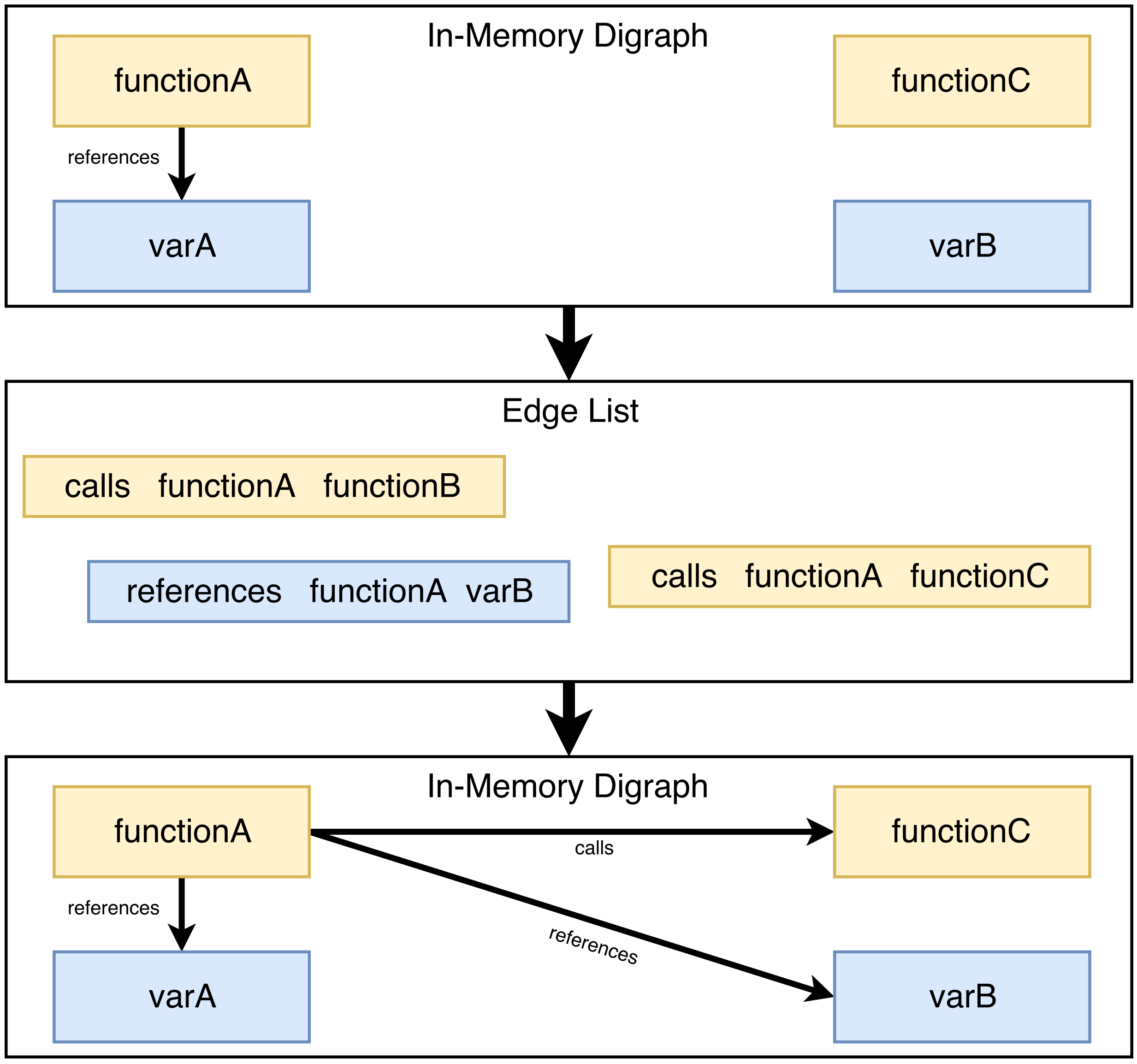


Figure 3.5: An example of the edge list data structure.

carrying out this conversion. This algorithm starts with a set of nodes called N and a set of edges called E. First, before the TA facts can be printed, a predefined schema needs to be printed that the graph conforms to. This schema is hardcoded and does not change between runs of the algorithm. Once the schema is printed, 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>`. Since forward declaration is not allowed in TA, all nodes are declared in the TA file before all edges. Last, 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. Once done, the outputted TA file is equivalent to the in-memory digraph.

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. 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 components. 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 and ClangEx are extremely similar which highlights the 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.

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
```

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 compilable units. The `compContains` relation connects `components` with any `class` located inside that component.

The `class` item is the next level of the model. Since this is where specific C++ language features and ROS elements might be defined, `class` nodes might contain `functions`, `variables` or ROS components such as `publishers` and `subscribers`.

The `function` node describes functions declared and defined in C++ source code. These functions have several C++-related attributes such as *isStatic*, *isVolatile*, and *isVariadic*. Further, functions also have a boolean flag called *isCallbackFunc* that indicates whether that function is a callback function for when a ROS subscriber receives data from a topic. Lastly, functions form a relationship with each other that indicates the functions that call other functions.

The `variable` node describes local variables or class fields. These nodes have several different attributes such as *isStatic*, *isPublisher*, and *isSubscriber*. Additionally if a variable is used in some sort of control structure, be it a loop or if statement, a flag called *isControlFlow* is set. Variables form relationships with several other entities in the metamodel. First, they can be contained in classes (if they are fields) or functions (if they are local). Next, if a function reads or writes to a variable, a relationship called `reads` or `writes` is formed between the function and variable. Last, flow of data between variables can be traced with the *varWrites* relation. If one variable writes its data into another variable, a *varWrites* relation will be added.

The `publisher` and `subscriber` nodes are ROS features that describe when a class communicates with some ROS topic. Publisher 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. Then, whenever `publish` call is made, a link is established between that publisher and the topic. Whenever a `subscribe` call is encountered, a subscriber node is created and then a link is created between it an a topic. In addition, if that subscriber uses a callback function, a link is created between it and the callback function.

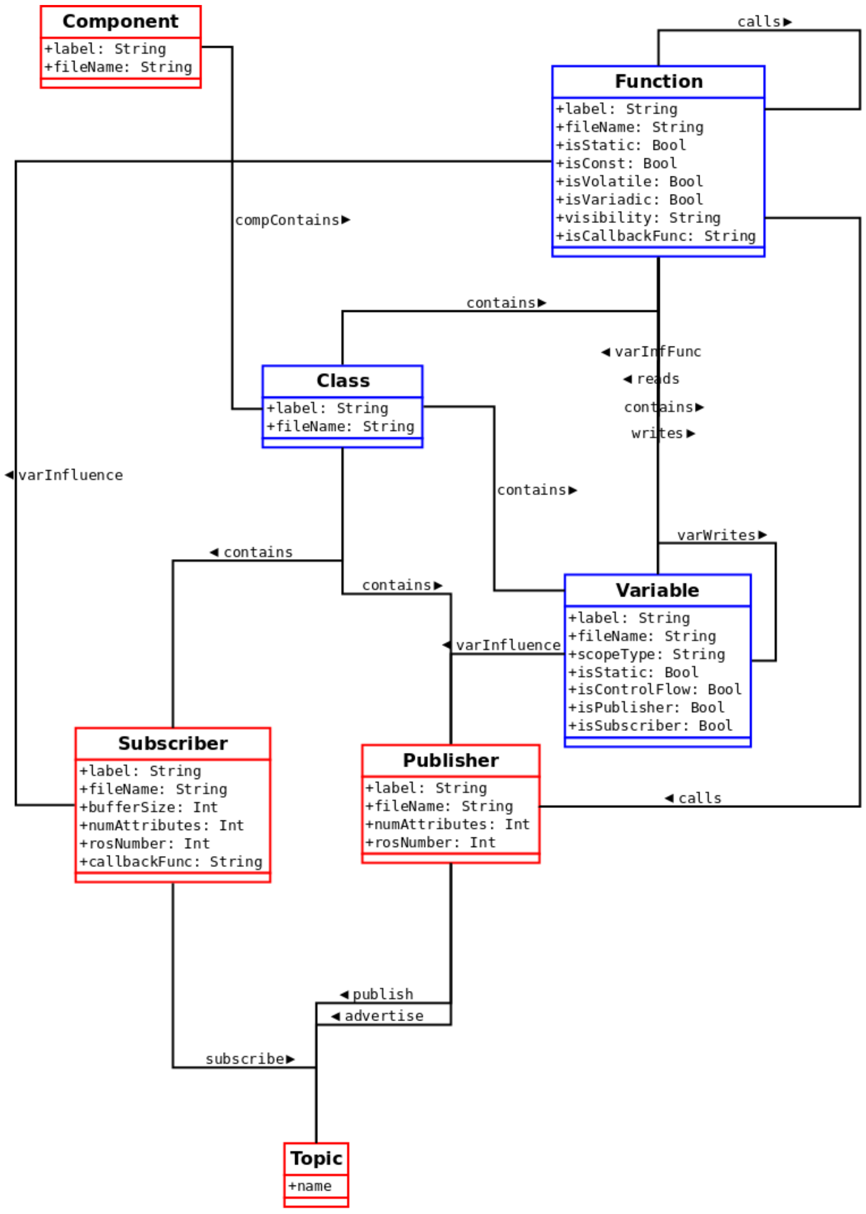


Figure 3.6: The Rex metamodel for full-analysis mode. Red classes are ROS features and blue classes are C++ features.

3.2.2 Advantages of the Rex Extractor

Since 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. Further, while Rex only targets ROS-based projects, insights that were gained from building Rex could be used to explore the development of new extractors that target different messaging frameworks. These new extractors could support messages sent between components connected by a bus or a network connection (Ethernet or CAN). While developing these new extractors requires much future work, solutions to challenges that were encountered during the development of Rex could potentially be recycled during extractor development for these platforms.

3.2.3 Disadvantages of the Rex Extractor

With Rex being built upon ClangEx, many of the disadvantages that exist in the ClangEx extractor are also present in Rex. Issues including its being bound to the Clang compiler and requiring the use of compilation databases exist. 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. While it is also possible for ClangEx to generate incorrect facts, Rex suffers more from this issue due to how ROS topics are created. Since Rex extracts ROS topics by looking at function calls to the `advertise` and `subscribe` ROS library functions, Rex can be fooled into adding incorrect topics or relationships into the final model. If one component creates a topic by passing a string literal as the parameter for the name of the topic while another component passes a string variable as the parameter for the name of the topic, Rex will see that as two different topics being created even if the two strings are equivalent. As such, before Rex can be run, a user has to check all the publisher and subscriber code to ensure string literals are being used for the topic name parameter in `advertise` and `subscribe` calls[5].

[5]For more information on this, see Figure 6.1.

3.2.4 Rex Internals

The implementation of Rex is very similar in make-up to that of ClangEx. Rex uses Clang's LibTooling API and AST Walkers to analyz 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. Whereas AST matchers would pass control over to Rex if a certain AST situation was met, Clang's AST visitor API allows for Rex to run code every time an AST node encountered. Due to this, Rex programmatically uses its own checks to look for particular situations such as calls to `publish` or `subscribe` to ROS topics. The reason the AST visitor API was used is because there are many different types of code fragments that could correspond to publishing or subscribing data to a topic. Further, since Clang does not have AST matchers that find specific ROS code fragments (like `advertise` or `subscribe` calls), using AST matchers would still require Rex to further narrow down these matches to only select `advertise` and `subscribe` calls. AST matchers were used in ClangEx since that extractor was just looking for simple C and C++ language features and not the type of items declared or used inside statements.

Other than these differences, Rex uses similar modules to ClangEx in a similar manner. Like ClangEx, there are three modules in use: (i) the Driver module that carries out command line operations and activates Clang; (ii) AST Walker modules that carry out how the AST is processed; and (iii) the Graph module that maintains an in-memory graph of the source code being analyzed. Since Rex uses similar models 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 areas of the AST in a particular source file, Rex runs each time a 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 break down that AST node into a declaration or statement of a specific type. For instance, declarations could be `variables`, `classes`, or `functions` and statements could be a call expression, binary/unary expression, or constructor call expression. For each of these subtypes, Rex has specific code that will run to adds corresponding nodes and edges to the digraph in the Graph module. For example, detecting a call expression statement between two functions would add a `calls` relation to the digraph where the source node is the caller function and the destination node is the callee function.

In addition to these changes, the Rex AST Walker has some special features for it to generate a model for detecting feature interaction hotspots. These include recording facts about `publishers`, `subscribers`, and variable reads and writes. The difficulty here is that, unlike C and C++ language features, Clang does not have built-in detectors that are capable of finding these ROS components. As such, the AST Walker has custom `publisher` and `subscriber` detectors. These detectors look in several different areas of the AST based on how the ROS communication API is used. Publisher and subscriber nodes are created by looking 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. Additionally, since these function calls create a publisher or subscriber objects, Rex has to remember the last publisher or subscriber created and then look for the variable that is going to store that publisher or subscriber object.

Creating `topic` nodes is a little more complicated since topics are not created through their own function calls. When a publisher or subscriber is created, one of the parameters is a string that is used to name the topic that 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 name of the variable or value of string literal obtained from the advertise/subscribe function call. Unfortunately, this presents a problem. Since Rex relies on the value of a parameter to create a topic node, the type passed to that parameter is important. If a variable is passed, Rex cannot resolve the value of that variable due to static analysis limitations. Instead, Rex will just use the variable name. That means that if the variable names containing the topic name given to two separate advertise calls re different between components (even if the topic name contained in that variable is the same), Rex will create two different topic nodes for the same topic.

The AST walker also has a variable read/write detection system that is used to determine whether a function reads or writes to a variable. This system 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. Since expressions can be composed of numerous sub-expressions, this algorithm starts at a top level expression and breaks the expression into each sub-expression recursively.

The algorithm is invoked by passing in a target expression, 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, since an expression either is binary, unary, or a variable[6], the algorithm then checks what type of expression the target expression is. If the expression is binary (meaning that the expression has a left-hand and right-hand side and a central operator), the algorithm gets the left and right sub-expressions as well as the central operator. Once done, it 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 subexpression is a variable, it simply adds that variable and the access type to the $varMap$ hash map. If the subexpression is another binary or unary expression, the algorithm calls itself with the LHS or RHS subexpression, 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 subexpression 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's an expression, the algorithm recursively calls itself with the subexpression, $varMap$, 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. Since a declaration reference is a reference to a `variable`, the algorithm records that $variable1$ was written to.

[6]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.

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:     else if $expr$ isa $UnaryExpression$ then
24:         $operator \leftarrow expr$.getOperator()
25:         $baseExpr \leftarrow expr$.getBase()
26:
27:         $currentVal \leftarrow$ getReadOrWriteUnary($upperVal$, $operator$)
28:         if $baseExpr$ isa $DeclRef$ then
29:             $varMap$.add($baseExpr$, $currentVal$)
30:             return $varMap$
31:         else
32:             return readWriteDetection($baseExpr$, $varMap$, $currentVal$)
```

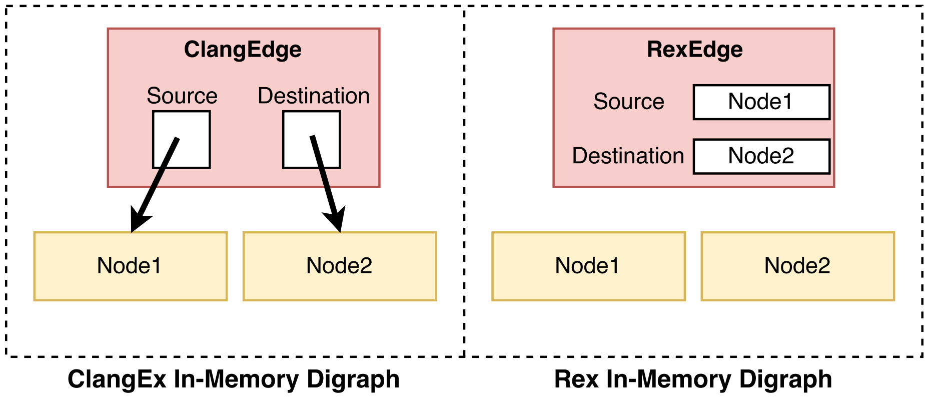


Figure 3.7: A comparison of how graphs in ClangEx and Rex are stored.

On the RHS of $variable1 = (variable2+ = variable1)$, the algorithm records that in the parent expression, the RHS side was read from. Here, since $(variable2+ = variable1)$ 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 $variable2$ and RHS being $variable1$. Since 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). Since both the LHS and RHS 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 and 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 digraphs slightly different. 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 that edge describes. 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 has to be resolved at the end of the program run (like in ClangEx), RexEdges just store the ID of the source and destination node instead of a pointer to the RexNode. Then, 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. Further, even though Rex has to check if the source and destination nodes exist before outputting the edges, ClangEx also has to do this.

Figure 3.7 shows an example of how Rex edges and ClangEx edges are stored in memory. For ClangEx, edges use pointers to refer to nodes that participate in relations. For Rex, edges use strings to store unique IDs of nodes that participate in relations.

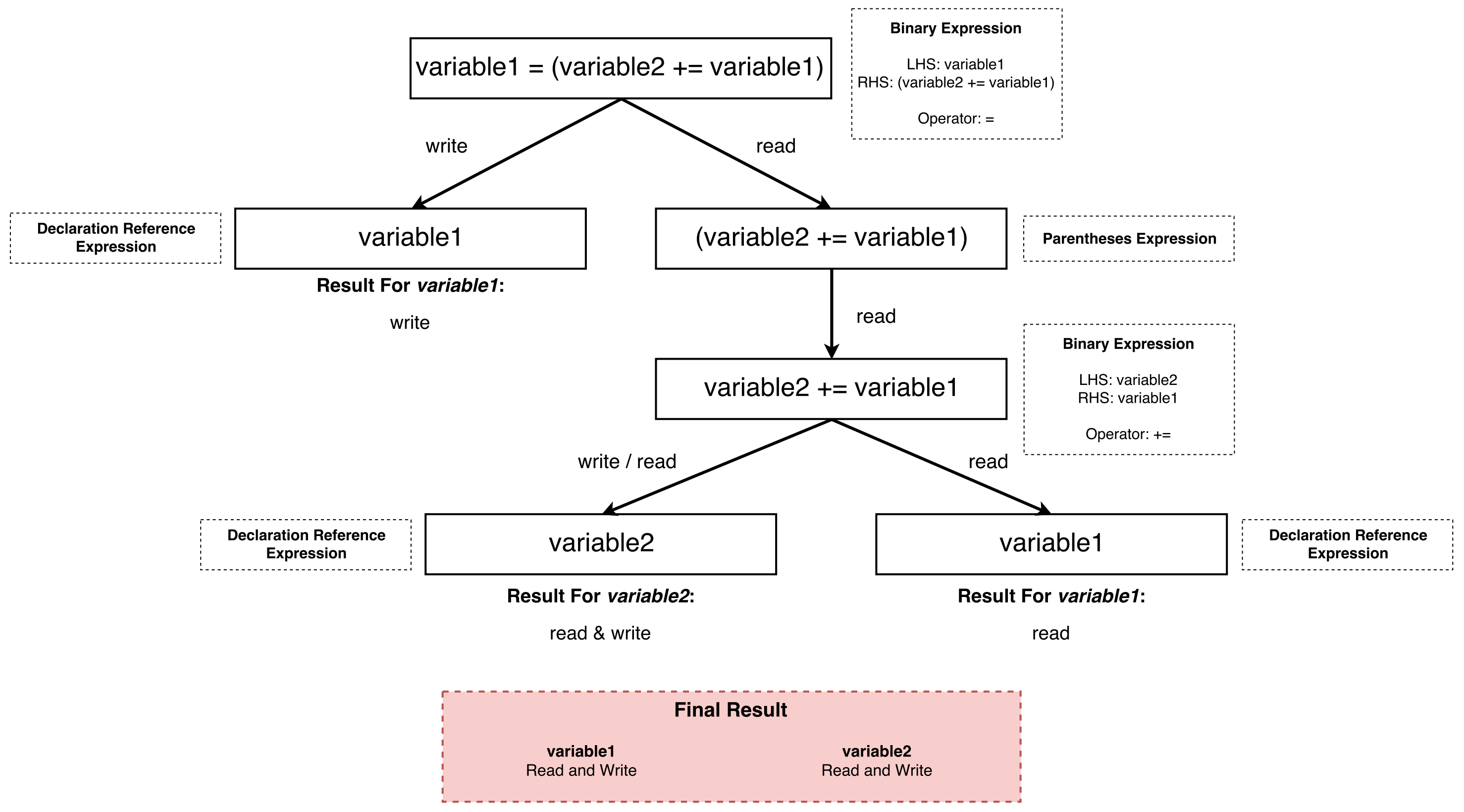


Figure 3.8: An example of the variable read/write detection algorithm.

Chapter 4

Analysis of Automotive Software

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 genericity 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. As a result, a fact extractor and Grok scripts need to be developed so that this message passing information is appropriately captured and analyzed.

This section describes how the relational algebra toolchain can be used to analyze automotive software to detect potential feature interactions. Section 4.1 presents the unique characteristics of automotive software and highlights several inherent challenges to analyzing this software. Section 4.2 introduces several different classes of feature interactions hotspots that might be present in automotive software.

4.1 Automotive Architecture

From a high-level, automotive software can be characterized as a collection of features that each perform a specific task in the vehicle. The difference between automotive systems and traditional monolithic systems is that automotive systems isolate and distribute these features across numerous embedded electronic control units (ECUs). Each of these ECUs might be responsible for a number of different features, are capable of receiving input from sensors, and can act on vehicle actuators. An example of a common vehicle ECU is the Electronic Control Braking Module (EBCM). The EBCM is responsible for implementing the antilock braking (ABS) and electronic stability control (ESC) features [54]. To implement these two features, it needs to be able to take input from sensors and other ECUs and apply the brakes in certain situations. The modern car has hundreds of such ECUs that are each responsible for implementing countless other features that can affect consumer safety [4]. The remainder of this section describes the method in which ECUs communicate and then discusses why this architecture can be challenging to analyze statically.

4.1.1 CAN Bus Protocol

To communicate with each other, features rely on a centralized vehicle bus known as the Controller Area Network (CAN). First introduced by Bosch in 1986, this bus is used in the majority of modern automobiles due to its high-speed data rate, error detection features, and data consistency for all nodes in the network [55]. Figure 4.1 shows how the CAN bus protocol connects ECUs in an automotive system. The bus consists of two signal lines called CANH and CANL which make up a 120Ω twisted pair wire. Nodes can be connected in any order and consist of an ECU and transceiver that pushes data onto the bus. This protocol dictates that communication be generic; there may or may not be a master communication node and nodes can be added or removed during operation of the system.

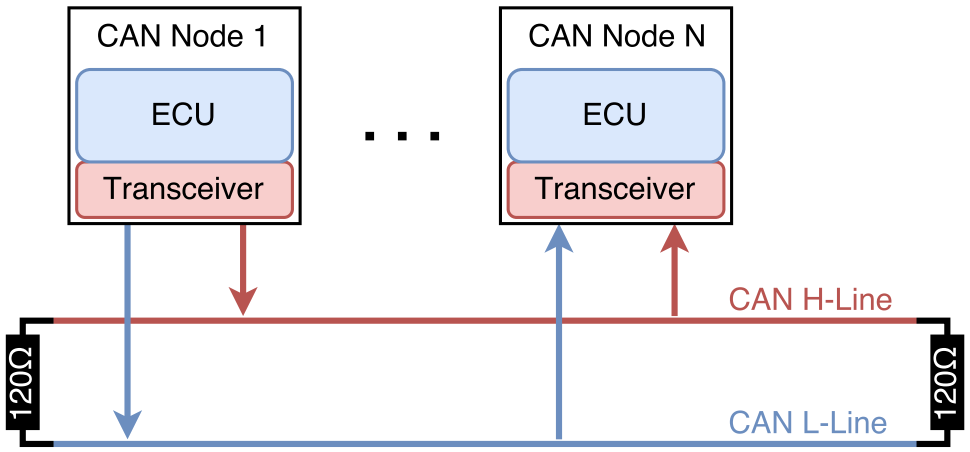


Figure 4.1: A CAN bus with two connected ECUs.

The CAN bus uses a broadcast protocol. Data sent from one node onto the bus is "heard" by all other nodes. Although messages are global, nodes can decide which messages they want to keep based on a filter that is applied to each message [56]. Messages sent over the bus could be processed by zero, one, or many other nodes.

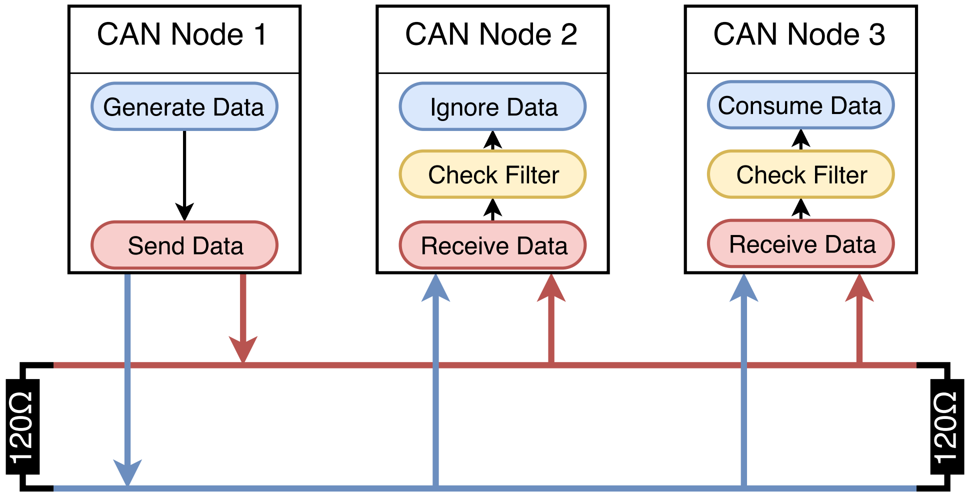


Figure 4.2: An example where data is sent on the CAN bus.

Figure 4.2 shows an example of how data propagates across the CAN bus. In this figure, CAN node 1 pushes data onto the bus. Since this protocol is global, CAN nodes 2 and 3 both receive the message and apply their filters on it. CAN node 3 keeps the message since its filter accepts it while CAN node 2 discards the message since its filter rejects it.

4.1.2 Challenges with Analysis

Due to the unique architecture of automotive systems, there are several inherent challenges that exist when analyzing this software statically. Since the goal of detecting hotspots is to identify areas of potential feature interactions, it is imperative that models representing the underlying software system need to be accurate and queries written to detect these hotspots need to cover all possible ways that hotspot might present itself. Further, with automotive software being safety-critical, this approach should attempt to reduce the number of false negatives detected.

Importantly, feature interaction hotspots need to focus on the communications between features rather than interactions at the function or variable level. As such, fact extractors that target automotive software must be able to accurately detect message-passing information. The difficult of extracting this information varies depending on the communication architecture used; for the ROS framework this is easy since components `publish` and `subscribe` to a named bus using a standardized one-line function call. For the CAN bus architecture, detecting message-passing has the potential of being quite difficult since CAN controller APIs are not standardized and differ in complexity. Building an extractor that can handle the code designed for multiple CAN bus controllers would be impossible unless it was known which controllers an automotive company used before the extractor was built.

Luckily these problems can be resolved. While CAN controllers are not standardized, developing a general-purpose automotive extractor might be possible through the AUTOSAR framework. This framework is partnership amongst all major automotive companies and aims to develop a common software framework for ECUs [58]. AUTOSAR divides the software on the ECU into three different layers; the application, runtime, and service layers. This division is beneficial as a fact extractor could be built to target "higher levels" of the stack. This is especially important for extracting communication information; AUTOSAR abstracts ECU communications as a full communication stack [59]. This stack allows for developers to use high-level code segments to communicate with other ECUs through simple function calls. This is similar to the `publish` and `subscribe` function calls used in ROS systems. With this abstraction, a fact extractor could be built that only looks at the application layer of an automotive system and detect communications of features.

4.2 Hotspots in Automotive Systems

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 vary depending on the 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 4.3 provides a breakdown of each of these three hotspot classes. Each of these classes contain several different hotspots that each identify a specific issue in the software project. Feature communication hotspots show the flow of information amongst the features in the project. Multiple input hotspots identify communications that flow from multiple features to a single feature and that could result in race conditions. Lastly, control flow hotspots identify communications between features where the message might affect the behaviour of the destination feature.

The remainder of this section introduces each class of hotspots and gives a definition

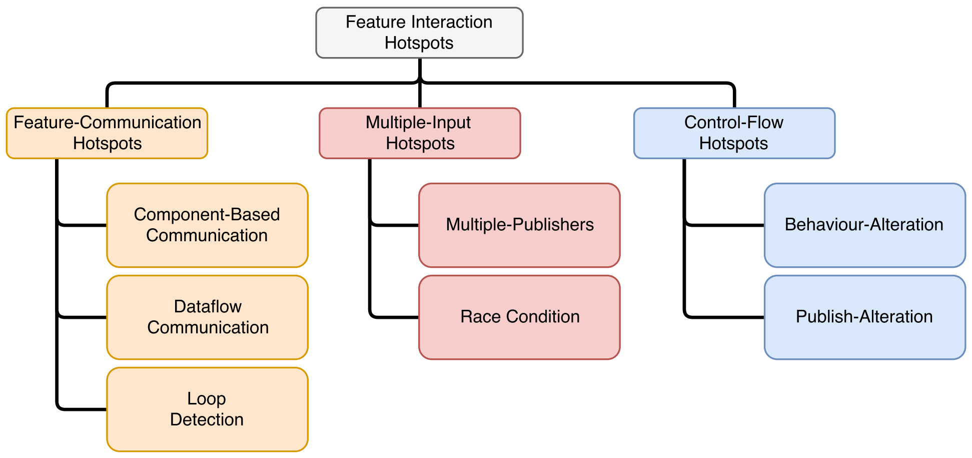


Figure 4.3: The breakdown of feature interaction hotspots.

of the hotspots that are part of each class. Section 4.2.1 describes feature communication hotspots. Section 4.2.2 describes multiple input hotspots. Section 4.2.3 describes control flow hotspots.

4.2.1 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 [60], a developer might not know precisely which other features their feature communicates with indirectly or might not be aware that their feature messages itself indirectly.

There are three main hotspots in this class: *component-based communication*, *dataflow communication*, and *loop detection*. Table 4.1 provides an overview for each of these hotspots. The remainder of this section describes each hotspot in detail.

Component-Based Communication

The component-based communication hotspot identifies features that send messages to other features. This hotspot detects the origin and destination feature where a communi-

Hotspot Name	*Description*	*Relation*
Component-Based Communication (CBC)	Describes features that message other features directly or indirectly. Only looks at the high-level communications.	**Direct:** $CBC \equiv DF$ **Indirect:** $CBC \supseteq DF$
Dataflow Communication (DF)	Describes features that message other features directly or indirectly. For indirect messages, looks at function calls in between.	**Direct:** $DF \equiv CBC$ **Indirect:** $DF \subseteq CBC$
Loop Detection (LD)	Describes communication loops where a feature messages itself directly or indirectly. Uses the component-based and dataflow methodologies.	Uses CBC and DF hotspots in detecting loops.

Table 4.1: The three hotspots part of the feature communication class.

cation takes place. Importantly, this hotspot detects both direct and indirect feature communications. Direct feature communication occurs when a source feature directly sends a message to the destination feature. Indirect feature communication is where a source feature sends a message that flows through a series of intermediate features. Eventually, the destination feature will receive a message This message will not be the original message sent out by the origin feature but may have been spawned due to the origin feature's message.

Detecting this hotspot is advantageous for several reasons. First, it allows for developers to gain better insight into which features in the automotive system interact meaning that developers can use it to see which features their feature indirectly messages. Additionally, this hotspot has the potential to identify how changes in one feature might propagate to other features.

Figure 4.4 shows an example of both the direct and indirect component-based hotspot. In this example *feature A* communicates with *feature B*. In the direct case, feature A sends a direct message to feature B. In the indirect case, feature A sends a message to feature x which sends a message to another set of unspecified features before a message is finally received by feature B. This set of unspecified features can consist of zero or more unrelated features.

Detecting this hotspot requires a TA model with little elements. It requires: (i) feature entities; and (ii) a relation called `direct` that describes which features directly message

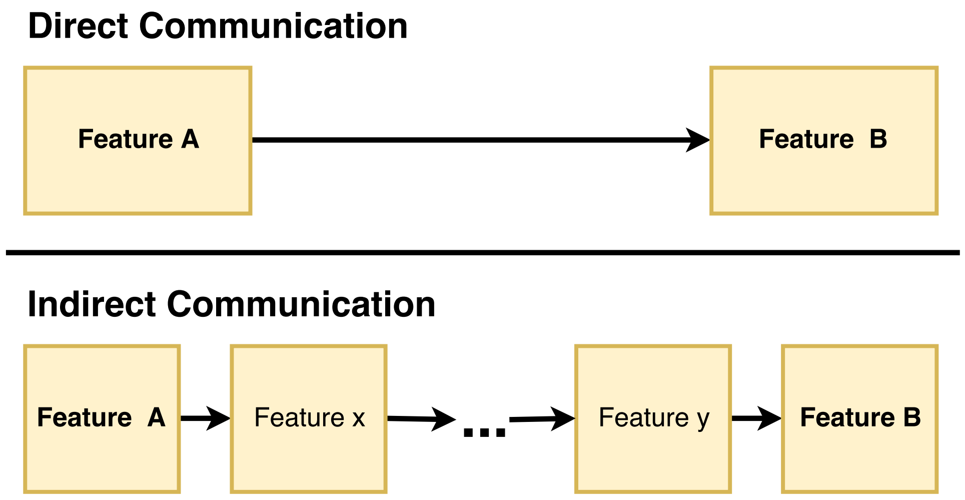


Figure 4.4: The difference between direct and indirect communications between features A and B.

other features. Given this model, this hotspot can be detected using relational algebra as follows:

1. For direct messages, simply print the contents of the `direct` relation.
2. Compute the transitive closure of the `direct` relation and store it in a relation called `indirect`.
3. Remove any entries from `indirect` that are also in `direct` and store the results back in `indirect` (`indirect` = `indirect` − `direct`). Print the results.

These steps can vary depending on the fact extractor, model schema, and names of the relations being used.

Dataflow Communication

While detecting instances of component-based communication can be beneficial, it is a high-level hotspot that ignores the function calls inside each feature. The dataflow communication hotspot attempts to address this by examining messages between features at the function call level. This means that, for an instance of this hotspot to be detected, there has to be a continuous path of function calls from start feature to end feature. For direct feature messages, this hotspot produces the same results as detecting direct component-based instances. This is because, with direct communication, there is **always** a link between two features messaging each other. For indirect feature communication, detecting instances of the dataflow communication hotspot will generate fewer results compared to the component-based hotspot since some of the instances of the indirect component-based hotspot might not have a continuous function call link from start to finish.

Figure 4.5 compares the component-based communication and dataflow communication hotspots in an example project. Here, there are three features called `A`, `B`, and `C` each with functions defined inside them. Looking at the direct communications instances, notice that the results for the component-based and dataflow hotspots are the same. However, when looking at the indirect results, note that there is one indirect component-based instance and zero indirect dataflow instances. The reason this is not considered to be an indirect dataflow hotspot is because there is no function call between the `receiveAData` and `sendBData` functions.

There are advantages and disadvantages with detecting this hotspot. The major advantage is that it gives users more detailed results with respect to how data flows inside features. While this hotspot may appear to be better than the component-based communication hotspot, it may also provide false negatives. For instance, consider Figure 4.5 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, the dataflow hotspot would not detect this interaction.

Detecting this hotspot requires a model that has several different entities and relations. A TA model needs to contain: (i) feature entities; (ii) function entities; (iii) a relation called `direct` that describes which features directly message other features; and (iv) a relation called `call` that describes which functions call other functions. Then, with that model, this hotspot can be detected using the following steps:

1. For direct messages, like the component-based communication hotspot, simply print the contents of the `direct` relation.

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 `communicate` ($\texttt{communicate} = \texttt{direct} + \texttt{call}$).

3. Compute the transitive closure of `communicate`. Remove any entries from this relation that do not start with the last function in a feature and end with the first function

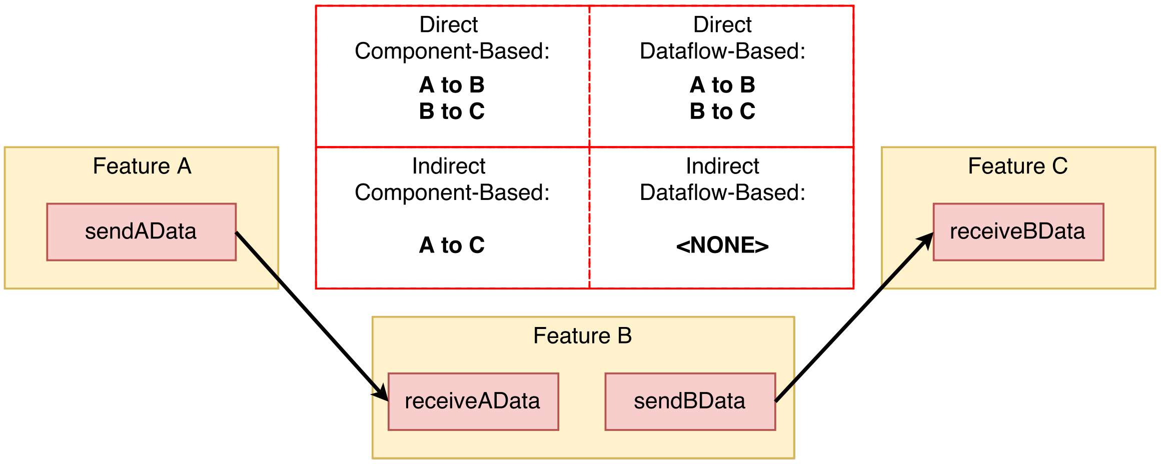


Figure 4.5: The difference between the component-based and dataflow communication hotspots.

in a feature. This is done to ensure the flow of data in a class between subscribe and publish calls is connected. Store these results in a relation called **indirect**.

4. Remove any entries from `indirect` that are also in `direct` and store the results back in `indirect` (`indirect` = `indirect` − `direct`). Print the results.

Loop Detection

Loop detection is the final 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 path of messages that flow from some origin feature back to itself. Loops can be detected using the component-based or dataflow-based detection methodology. For component-based loops, only the messages sent between functions are considered whereas for dataflow-based loops, the messages and function calls are both considered. As before, the set of direct loops detected using both methodologies will be the same. Only the indirect loop results will differ.

The benefit of this hotspot is that it can uncover unexpected self-loops in the communication graph. A detrimental self-loop could occur when a feature participates in a feedback loop where messages it receives from itself could amplify its output to other features. Additionally, since this hotspot detects both component-based and dataflow-based loops, another benefit is that users can choose the detection methodology that suits their needs. For instance, a user might only care about dataflow loops since it would indicate that there is continuous chain of function calls and messages from start to finish.

This hotspot is an extension of the first two hotspots in this class. Similar to the dataflow hotspot, a TA model needs to contain the following facts: (i) feature entities; (ii) function entities; (iii) a relation called `direct` that describes which features directly message other features; and (iv) a relation called `call` that describes which functions call other functions. With this model, component-based and dataflow-based loops can be detected using the following steps:

1. For direct component-based and dataflow-based loops, take the `direct` relation and remove relations from it where the domain and range are not the same. Print the results.

2. For indirect component-based loops, take the transitive closure of `direct` and store it in a relation `indirectCompLoops`. Then, remove relations from it where the domain and range are not the same. Print the results.

3. For indirect dataflow 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 `communicate`.

4. Compute the transitive closure of `communicate`, remove entries that do not start with the last function in a feature and end with the first function in a feature, and then determine the parent feature that each function belongs to. Store these results back in the `communicate` relation. Remove entries from this relation where the domain and range are not the same.

5. Print the `communicate` relations to get the indirect dataflow loops.

4.2.2 Multiple Input Hotspots

The multiple input hotspot class detects instances where multiple features send data to the same feature. These instances might result in a bug where a message from one feature is drowned out by another competing feature which sends a large number of messages rapidly. This hotspot class can also detect “race-condition”-like situations where data received from multiple features update a single variable. In this case, updates made to that variable from one feature could potentially be lost.

Hotspot Name	*Description*	*Relation*
Multiple Publishers (MP)	Describes situations where multiple features message the same feature through the same callback function	$MP \subseteq CBC$ $MP \subseteq DF$
Race Condition (RC)	Describes situations where multiple features message some feature causing updates to the same variable.	$RC \subseteq CBC$ $RC \subseteq DF$

Table 4.2: 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 4.2 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.

Multiple Publishers

The multiple publishers hotspot identifies situations where multiple features message the same function inside a feature. Since this hotspot can manifest itself differently depending on the communication architecture used, any implementations to detect this hotspot needs to take this into account. For instance, in ROS, features publish data to topics which are then sent to features that subscribe to these topics. Therefore, an implementation to detect this hotspot in ROS would need to detect situations where multiple publishers send data to the same topic and where multiple topics send data to the same callback function.

The benefit of detecting this hotspot is that it can uncover situations where multiple features send data to a single recipient function causing one feature to overpower the other functions. In an architecture like ROS, this is especially prevalent since each feature has an incoming and outgoing message buffer for each topic it subscribes and publishes to. In this case, if one feature floods this queue, messages from other features are dropped.

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: (i) feature entities; (ii) function entities; (iii) a relation called `direct` that describes which features directly message other features; and (iv) a relation called

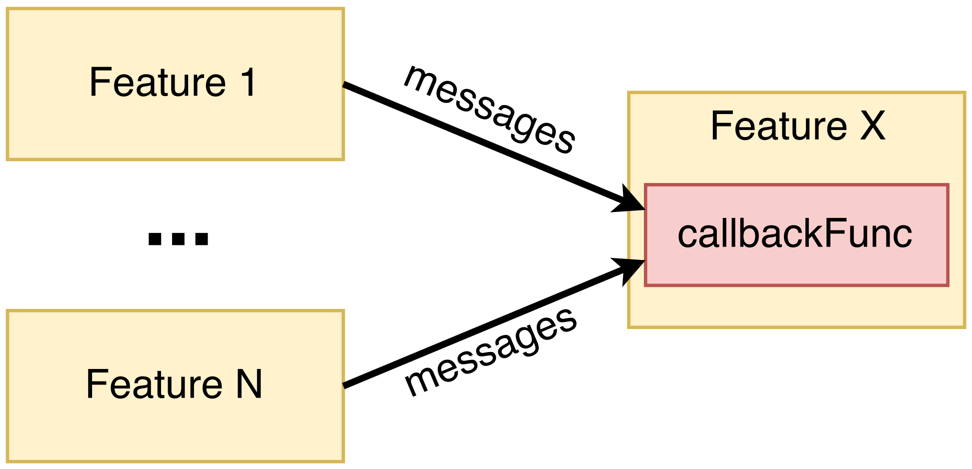


Figure 4.6: An example of the multiple publisher hotspot.

`call` that describes which functions call other functions. The model used to detect this hotspot does not have to be detailed because this hotspot is only concerned with calls between features. To detect this hotspot the following relational algebra calls are required:

1. For each callback function in the project, take the `direct` relation and apply the callback function to the range. Store the results in a new relation called `instances`.
2. Get the number of entries in the `instances` relation. If there are 1 or less, go back to the first step for the next callback function.
3. Get the parent feature name of that callback function and print it. Then, print the `instances` relation. This gives all the features that message that callback function. Then return to step one for the next callback function.

Race Condition

The race condition hotspot describes situations where multiple features send messages that end up modifying the same variable. Since 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.

Detecting this hotspot is important as it has the potential of highlighting race conditions in a feature. With message-passing systems, there is the possibility that a variable can be modified by two different features separately through different channels. Modifications made to this variable by one feature might be overwritten by other features prior to that variable being used. For instance, say there is a variable that keeps track of the X and Y positions of a robot and two features are both updating this variable. A program might run into a lost update scenario if both features modify this variable before the X and Y positions are used.

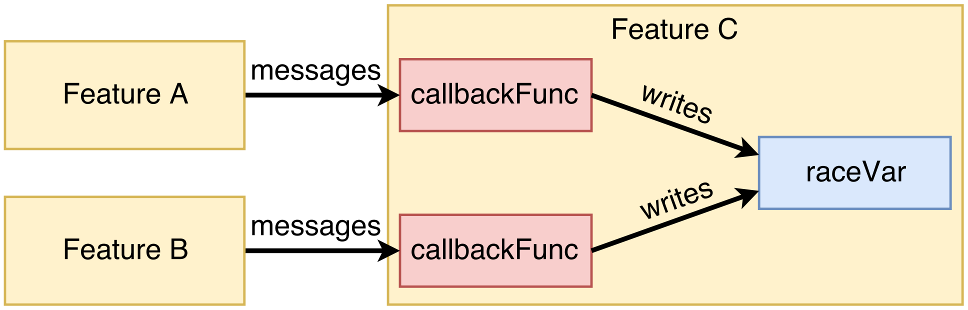


Figure 4.7: An example of the race condition hotspot.

Figure 4.7 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 race condition hotspots using relational algebra requires a model that is a little bit more detailed compared to the one used for the multiple publishers hotspot. This is because detecting this hotspot also requires knowledge of variables being written to. As such, the following is required: (i) feature entities; (ii) function entities; (iii) **variable entities**; (iv) a relation called `direct` that describes which features directly message other features; (v) a relation called `call` that describes which functions call which other functions; and (vi) **a relation called `write` describing which functions write to which variables**. The bold text indicates the differences in the model between this and the multiple publisher hotspot model. To detect race conditions, the following relational algebra calls are required:

1. For the `write` relation, keep only cases involving a callback function writing to a variable. Store this in a new relation called `callbackWrite`. This now shows the variables that all callback function writes to.

2. Then, for each variable in the range of `callbackWrite`, get the number of callback functions that write to it. If more than one callback function writes to it, proceed to the next step for that variable.

3. For that variable, get the feature that it belongs to and print it. Then, for each callback function that writes to it, print the features that message that callback function.

4.2.3 Control Flow Hotspots

Another class of hotspots that can be problematic are *control flow* hotspots. These hotspots describe situations when the behaviour of one feature is altered due to messages received 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. This allows developers to focus on interactions between features that have a definitive effect on that feature's behaviour.

Hotspot Name	*Description*	*Relation*
Behaviour Alteration (BA)	Describes situations where messages from a feature cause that feature to alter how it behaves.	$BA \subseteq CBC$ $BA \subseteq DF$ $BA \supseteq PA$
Publish Alteration (PA)	Describes situations where messages from a feature cause that feature to send messages to other features.	$PA \subseteq CBC$ $PA \subseteq DF$ $PA \subseteq BA$

Table 4.3: 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 4.3 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.

Behaviour Alteration

The behaviour alteration hotspot describes situations where a message from one feature causes another feature to change its behaviour. The behaviour of a feature is considered changed when a variable that is written to by a callback function or a function called directly or indirectly by that callback function is eventually used in the decision block 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 transfer their data into other variables that then get used in control structures.

Detecting this hotspot is important because knowing which communications change the behaviour of a feature is important. This hotspot provides even greater detail then feature communication hotspots since it examines how features operate when messaged by other features. With the results from this hotspot, developers can place further care in ensuring these communications are handled properly.

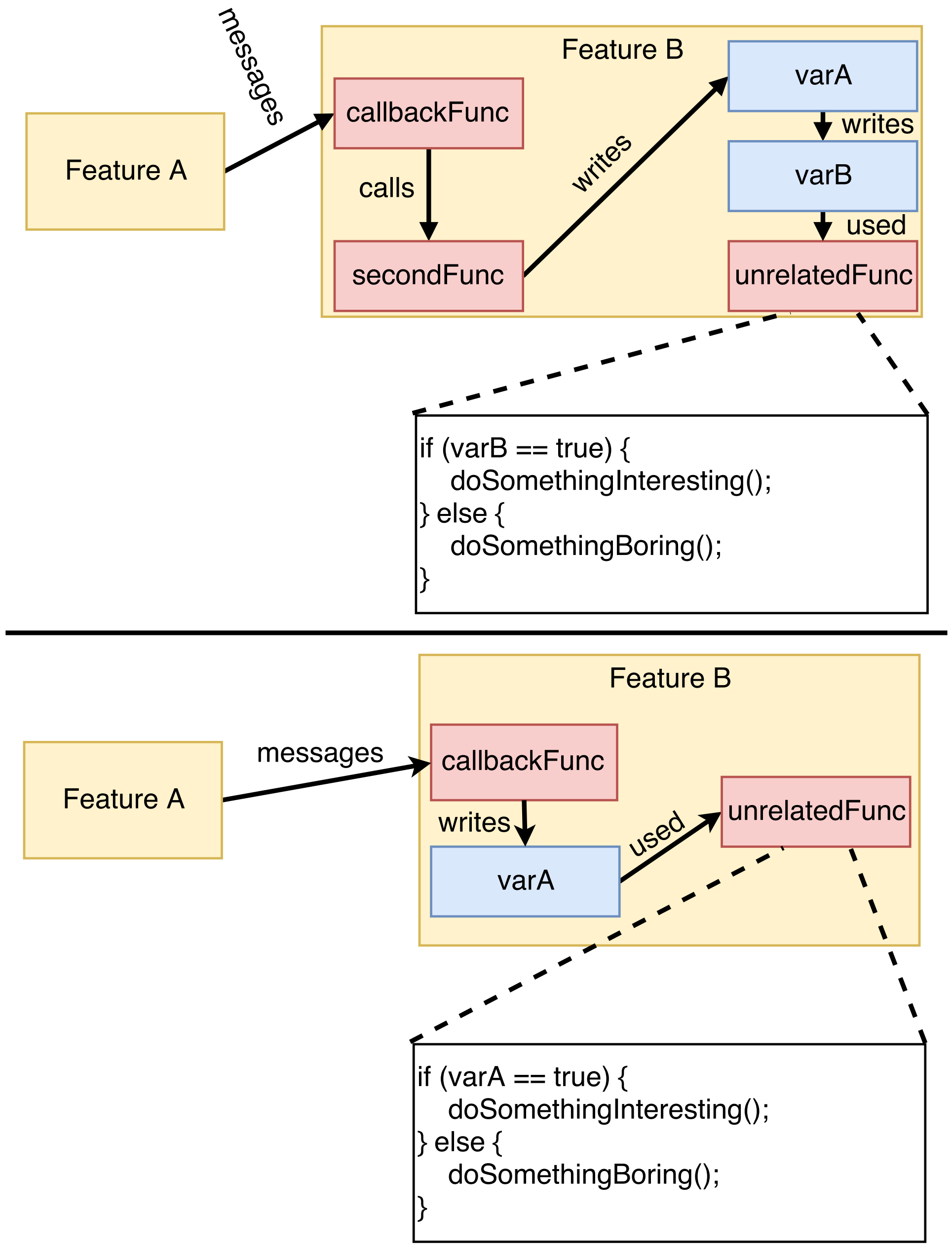
Figure 4.8 shows two different instances of the behaviour alteration hotspot. These two instances are not exhaustive since there are numerous ways this hotspot manifests itself. As before, yellow boxes are features, red boxes are functions, and blue boxes are variables. The first configuration is fairly complex; feature A messages the `callbackFunc` function in feature B which then calls the `secondFunc` function. Inside `secondFunc`, `varA` is written to. Then, that variable writes data to another variable called `varB`. Lastly, `varB` is used in an `if` statement inside `unrelatedFunc`. The second instance occurs when feature A messages feature B causing a variable called `varA` to be modified. This variable is then used in another function called `unrelatedFunc` in an `if` statement. From both these examples, feature B's behaviour is being modified as a result of a message from feature A.

Detecting this hotspot using relational algebra requires a fairly detailed model that includes: (i) feature entities; (ii) function entities; (iii) variable entities; (iv) a relation called `direct` that describes which features directly message other features; (v) a relation called `call` that describes which functions call which other functions; (vi) a relation called `write` that describes which functions write to which variables; (vii) a relation called `varWrite` that describes which variables write to which other variables; (viii) an attribute called `isControlVar` which denotes whether a variable is part of a control flow structure; and (ix) an attribute called `isCallbackFunc` which denotes whether a function receives data from another feature. With a model containing this information, the behaviour alteration hotspot can detected using the following relational algebra calls:

1. First, create a set called `controlVars` that contains variables that are present in some control structure. This can be done by looking at the `isControlVar` attribute.

2. Create another set called `callbackFuncs` that contains functions that receive data from other features. This can be done by looking at the `isCallbackFunc` attribute.

3. Create a master relation called `masterCalls` that contains the union of the following relations: `call`, `write`, and `varWrite`. This relation contains every single instance where a function or a variable transfers data.



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

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

Figure 4.8: Two examples of the behaviour alteration hotspot with two features: A and B.

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.

5. Apply the `callbackFuncs` set on the domain of the `masterCalls` relation using relational selection. Store these results back in `masterCalls`. Now, every entry in the `masterCalls` relation starts with a callback function.

6. Last, apply the `controlVars` set on the range of the `masterCalls` relation using relational selection. Store these results in `masterCalls`. This relation now denotes which callback functions affect the feature's behaviour.

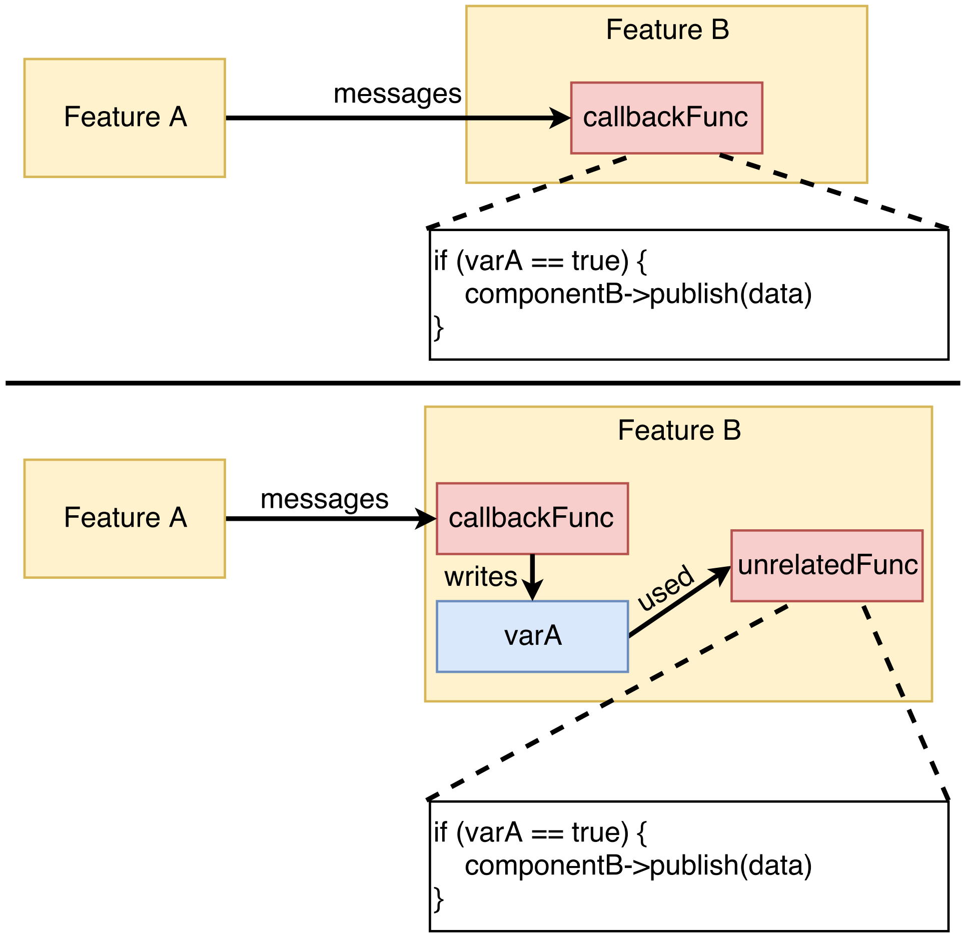
7. To get the features that directly affect the behaviour of other features, join the `direct` relation on the `masterCalls` relation. This shows which features message that callback function.

8. To get indirect instances, repeat the previous step but instead of applying `direct`, apply the transitive closure of `direct`.

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 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. While it might be easy to determine which features message each other, determining whether features affect how another feature communicates. For instance, while ROS provides a visualization tool called `rqt_graph` that dynamically shows which features message other features, there is no way



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

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

Figure 4.9: Two examples of the publish alteration hotspot for two features: A and B.

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 4.9 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 that includes: (i) feature entities; (ii) function entities; (iii) variable entities;

(iv) a relation called `direct` that describes which features directly message other features; and (v) a relation called `call` that describes which functions call which other functions; (vi) a relation called `write` that describes which functions write to which variables; (vii) a relation called `varWrite` that describes which variables write to which other variables; (viii) **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**; (ix) **a relation called `varFuncInf` 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**; (x) an attribute called `isControlVar` which denotes whether a variable is part of a control flow structure; and (xi) an attribute called `isCallbackFunc` which denotes whether a function receives data from another feature. The bold elements show additions to the model from the behaviour alteration hotspot. Given such a model exists, publish alteration hotspots can be detected through the following algebraic operations:

1. Create a set called `callbackFuncs` that contains functions that receive data from other features. This can be done by looking at the `isCallbackFunc` attribute.

2. Create a master relation called `masterCalls` that contains the union of the following relations: `call`, `write`, `varWrite`, `varInfluence`, and `varFuncInf`. This relation contains every single instance where a function or a variable transfers data or where a control structure is modified leading to another function call or publish message.

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.

4. Apply the `callbackFuncs` set on the domain of the `masterCalls` relation using relational selection. Store these results back in `masterCalls`. Now, every entry in the `masterCalls` relation starts with a callback function.

5. Last, get the relation of publish calls and apply it on the range of the `masterCalls` relation using relational selection. Store these results in `masterCalls`. This relation now denotes which callback functions cause a publication to occur.

6. To get the features that affect whether a feature publishes data, join the `direct` relation on the `masterCalls` relation. This shows which features message that callback function.

7. To get indirect instances, repeat the previous step but instead of applying `direct`, apply the transitive closure of `direct`.

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 which detect each type of hotspot described in Chapter 4. Each of these scripts were then run on a TA model of the Autonomoose software and the results were analyzed.

Before the case study results are presented, a brief description of the Autonomoose autonomous driving project is introduced. 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 both deliberately obfuscated.

5.1 Autonomoose

Developed by the University of Waterloo Centre for Automotive Research (WatCAR), Autonomoose is an autonomous driving platform developed on 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 [61]. For a vehicle to be at this level, it needs to be in control of all aspects of the dynamic driving task and will 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 [62].

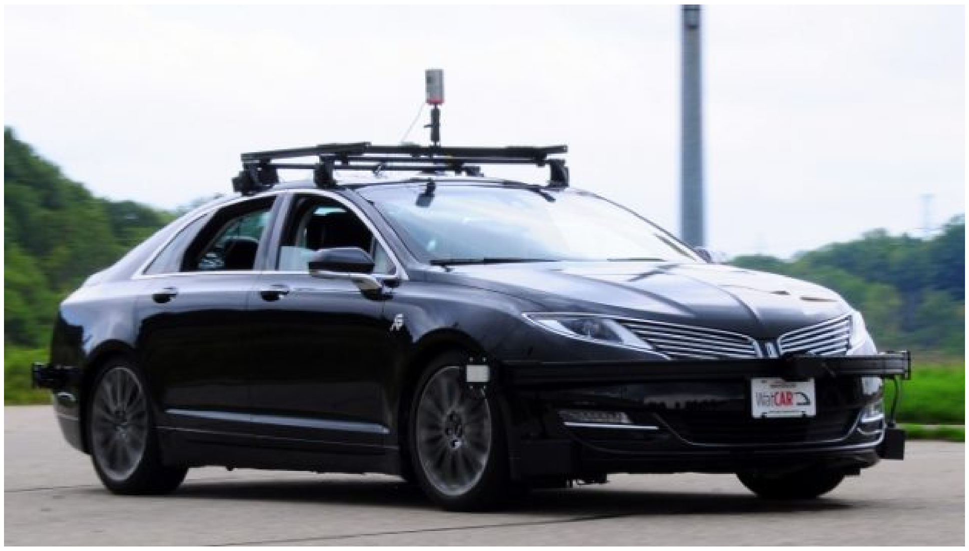


Figure 5.1: The modified Lincoln MKZ used by Autonomoose.

Similar to other autonomous driving projects, Lincoln MKZ used by Autonomoose is comprised of a plethora of sensors that allow it to drive autonomously. The vehicle features a global positioning system (GPS) and inertial measurement unit (IMU) that allow Autonomoose to discern the car's position on the road. The GPS has an accuracy of 5cm and updates at 20Hz while the IMU has a heading angle accuracy of 0.08 degrees and updates at 100Hz [63]. 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 built-in sensors to make decisions that get applied on 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. Rather than using the CAN bus, Autonomoose communicates using the ROS framework. Currently, the Autonomoose software stack operates on top of the vehicle's existing software stack. By using a ROS-CAN bridge, information obtained from vehicle sensors can be used by Autonomoose and commands from Autonomoose can be sent to vehicle actuators.

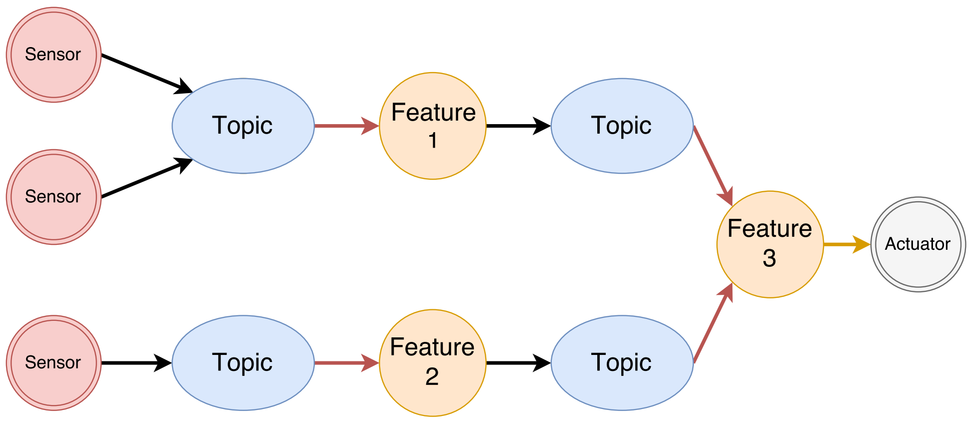


Figure 5.2: A high-level overview of the Autonomoose software stack.

While Autnomoose lacks a CAN bus architecture to send messages, it still beneficial to study it to similarities between the ROS and CAN frameworks. In systems that utilize a CAN bus for message-passing, nodes send messages across a shared bus. While these messages are received by all nodes, only nodes that have a filter that accept the message will consume it [64]. Similarly, in systems that use ROS, nodes send messages to topics which then get received by other nodes which find those topics pertinent [65]. Both of these architectures can result in the same type of communication configurations between nodes such as many-to-one and one-to-many. Further, similar to other modern vehicles, Autonomoose is comprised of a set of features that each communicate with one another to achieve a shared goal.

Lastly, the notion of features in Autnomoose needs to be discussed. As before, I define a feature as "a coherent and identifiable bundle of system functionality that helps characterize the system from a user perspective" [20]. In Autonomoose, features are each divided into separate ROS packages that each achieve a single goal. This configuration is beneficial for developers as it improves comprehension and improves relational analysis since features do not have to directly be delineated in the associated program models; a feature is simply a ROS package.

5.2 Autonomoose Case Study

As previously mentioned, a case study was conducted on Autonomoose to determine whether the relational algebra toolchain could be used to detect feature interaction hotspots.

As three different classes of feature interaction hotspots were proposed in Chapter 4, this case study tested each of the three classes. The classes tested are: (i) feature communication hotspots; (ii) multiple influence hotspots; and (iii) control flow hotspots.

To detect each of these hotspots, the entirety of the Autonomoose source code 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 via manual analysis to ensure completeness and correctness. As before, while this thesis presents aggregated results from this case study, models and direct results have been obfuscated due to confidentiality obligations with WatCAR 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 feature communication hotspots, Section 5.2.3 discusses multiple influence hotspots, and Section 5.2.4 discusses control flow hotspots. Lastly, Section 5.2.5 corroborates the results obtained with Autonomoose developers.

5.2.1 Setup

To detect hotspots in Autonomoose, some initial setup was required. First, I 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 5.1 shows the necessary entities and relations that need to be in an Autonomoose model to detect all hotspots. The necessity 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.

With Rex was developed, a model of the Autonomoose source code had to be generated. To do this, 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 since ROS-based projects use Catkin as a build tool and use a number of different compiler flags that are hidden to the user. Further, 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.

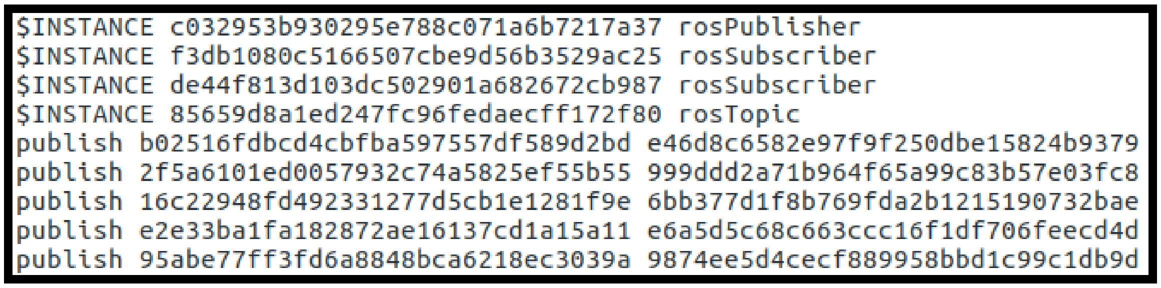
To solve this problem, I modified one of the top-level build scripts for Autonomoose to contain a command that forces Catkin to generate a compilation database for each ROS project compiled. Then, Autonomoose had to be cleanly built without Rex to generate this database. By doing this, Catkin created a `compile_commands.json` file for each feature

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 publishers are under which control structures. • A relation showing which functions are under which control structures.	• A boolean attribute denoting whether a variable participates in the decision portion of a control flow structure.

Table 5.1: Required model elements to detect all hotspots.

in Autonomoose. Since 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 Autonomoose's source code. To generate the models, all Autonomoose source files were added into Rex's processing queue and then alls "test" files were removed. Two different models were generated: `ANM_FULL.ta` and `ANM_MIN.ta`. The `ANM_MIN.ta` model is a lightweight model generated through Rex's simple-analysis mode. This version contains only ROS-based information, 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 and is extremely dense. These models were generated using a snapshot of the Autonomoose project from October of 2017. Figure 5.3 shows a portion of the minimal model used in this case study.



```
$INSTANCE c032953b930295e788c071a6b7217a37 rosPublisher
$INSTANCE f3db1080c5166507cbe9d56b3529ac25 rosSubscriber
$INSTANCE de44f813d103dc502901a682672cb987 rosSubscriber
$INSTANCE 85659d8a1ed247fc96fedaecff172f80 rosTopic
publish b02516fdbcd4cbfba597557df589d2bd e46d8c6582e97f9f250dbe15824b9379
publish 2f5a6101ed0057932c74a5825ef55b55 999ddd2a71b964f65a99c83b57e03fc8
publish 16c22948fd492331277d5cb1e1281f9e 6bb377d1f8b769fda2b1215190732bae
publish e2e33ba1fa182872ae16137cd1a15a11 e6a5d5c68c663ccc16f1df706feecd4d
publish 95abe77ff3fd6a8848bca6218ec3039a 9874ee5d4cecf889958bbd1c99c1db9d
```

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

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.

```
//Get a list of callback functions.
callbackFunc = rng (subscribe o call);

//Determines all the variables that are modified by each callback function.
vars = callbackFunc o write;
for curVar in rng vars {
  specific = vars . {curVar};

  //Gets the number of instances of that variable.
  if (#specific > 1) {
    //Deal with cases where multiple callbacks modify.
    print "For the " + curVar + " variable:";

    //Gets the callback functions that push to that variable.
    callbacks = dom specific;
    for cb in callbacks {
      //Prints the callback.
      {cb} . @label;
    }
  }
}
```

Figure 5.4: An example Grok script that detects the race condition hotspot.

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. As an example, Figure 5.4 shows the script that is capable of detecting all instances of the race detection hotspot in models extracted by Rex. Here, lines 2 and 5 get all the callback functions that write to variables. Then lines 6 through 21 iterate through each variable written to by callback functions to determine if two or more callback functions write to them.

5.2.2 Feature Communication Hotspots

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

Results

After running the three Grok hotspot scripts, the following results were obtained which are summarized in Table 5.2. For component-based communication, there are 12 instances of direct and indirect communications between features. For instances involving direct communication, only seven features in Autonomoose actually send messages to other features. For indirect messages, only four features in Autnomoose messages send messages indirectly to other features.

Hotspot Name	Direct Results	Indirect Results
Component-Based Communication	12	12
Dataflow Communication	12	3
Loop Detection	1	2

Table 5.2: Result of detecting feature communication hotspots in Autonomoose.

For cases of direct dataflow communications, the results are the exact same as the direct component-based messages since $DF = CBC$. For indirect dataflow communications, there are only three features that indirectly message other features. The reason there are few indirect instances despite Autnomoose's size, is because Autonomoose is not set up to forward messages when a feature receives a message. Rather, each time an Autonomoose feature spins and receives a message, it tends to change state which eventually causes it to publish a message to another feature. These type of instances are detected in the control flow hotspot class.

Lastly, for loop detection, there was one direct loop and two indirect component-based loops. No indirect dataflow-based loops were detected. This result was as expected since the dataflow communication hotspot found that there are only three instances of indirect dataflow communications and none of the instances were a loop. The most interesting result here is the presence of a direct loop. This means that a feature is sending data that directly loops back to itself. Since 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.

Verification

To detect feature communication hotspots, accurate information about: (i) `publishers`, (ii) `subscribers`, (iii) `topics`, (iv) `functions`, (v) `classes`, (vi) `publish` messages, (vii) `subscribe` messages, (viii) function `calls`, and (ix) hierarchies between `classes` and `functions` is needed in the resultant model. To ensure the completeness of the results, each of the ROS-based entities and relations in the model need to be verified for accuracy. To do this model entries need to be compared to the Autonomoose source code to ensure these entities and relations are correct. To perform this verification, a manual scan of the source code was conducted and a TA model containing manually extracted information was created. The entries in the Rex-generated model were compared to the manually-generated models to determine the model's completeness.

Element Type	**Source Code**	**TA Model**	**Accuracy**
Publisher Entities	52	52	100%
Topic Entities	74	71	95.9%
Subscriber Entities	46	46	100%
`Publish` Relation	60	60	100%
`Subscribe` Relation	46	46	100%

Table 5.3: Completeness of ROS-based entities and relations in the Autonomoose model.

Table 5.3 shows completeness statistics for all ROS-based elements in the Autonomoose model. For all but one entity, the Rex-generated TA model was exactly the same as the manually-generated model. The only entity with less than 100 percent accuracy is the `topic` entity as the Rex-generated model labels three topic names incorrectly. The danger with inaccuracies in the extracted topic name is that publisher and subscriber calls might not link up causing communications between features to be missed. In the case of the three invalid topics in the Autonomoose model, this occurred because the publishers using these topics created them using a variable containing the topic name while Rex is only able to parse topic names from string literals. This is a known limitation.

Other types of model elements needed to detect these hotspots such as function `calls` or `contain` hierarchies are assumed to be complete since they are parsed directly from abstract syntax tree using Clang. For instance, any function call present in source code will have a unique `callExpr` node in the AST.

Since communications between features is a characteristic common to all message-passing systems, verifying the correctness is not necessary. Ensuring the results are complete is enough for the purpose of this class of hotspots. Communications between features alone are not enough to indicate whether interactions between these features should be investigated further. The multiple input and control flow hotspots investigate their correctness with respect to the Autonomoose source code.

5.2.3 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 influence hotspots, the following results were obtained which are summarized in Table 5.4. First, there are no multiple publisher hotspots detected in the project. This means that any instance of communication between two features has its own dedicated communication channel and callback function. This result is as expected since designing the system in this manner reduces the risk of one feature overpowering another feature.

Hotspot Name	Results
Multiple Publishers	0
Race Condition	11

Table 5.4: Result of detecting multiple influence hotspots in Autonomoose.

For the race condition hotspot, there are 11 different variables that are affected by this hotspot. 8 of those 11 variables are located in the same component. Further, the number of callback functions that participate in the race condition varies depending 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.

Verification

To detect these hotspots, the following information is required: (i) `publishers`, (ii) `subscribers`, (iii) `topics`, (iv) `functions`, (v) `classes`, (vi) `publish` messages, (vii) `subscribe` messages, (viii) function `calls` (ix) **`variable writes`**, and (x) hierarchies between `classes` and `functions`. Information shown in bold was not verified in the feature communication section and will be checked for completeness here.

To check the `write` relation for completeness, a subset of all the functions in Autonomoose were selected and then each were scanned to manually find all cases where variables were written to. These manual cases were then compared to the Rex-generated Autonomoose model to check if the results compare. For this completeness check, the selected functions to analyze were all 44 callback functions that receive data from ROS topics. The reason these callback functions were chosen is that these functions tend to write to many variables as they are responsible for transferring data received from topics.

Element Type	**Source Code**	**TA Total**	**Correct**	**Incorrect**	**Accuracy**
`Write` Relation	287	226	226	0	78.75%

Table 5.5: Completeness of ROS-based entities required to detect multiple influence hotspots.

Table 5.5 presents the completeness results for the `write` relation. In the Rex-generated TA model, there are 226 cases where a callback function writes to a variable. Through manual analysis of all the callback functions, it was found that there are 287 cases where a variable is initialized or written to. Each of the results in the Rex-generated model were compared to manual analysis results and it was found that Rex correctly predicted 78.75% of all variable writes. While the accuracy is not high, it is still encouraging that all of Rex's predictions are correct. This means that all cases of variable writes in the Rex-generated Autonomoose model are present in the source code.

The reason the accuracy is around 75% is due to several limitations of Rex's read and write detector. First, developing an algorithm to statically detect variable accesses in C++ is impossible due to pointers and aliasing. In a situation where the address of one variable is written to a pointer, Rex would still consider that to be a write despite it truly being one. The second limitation is that Rex does not look at function calls that set internal object fields as a variable write. If an object is modified through the use of a function call (such as the `push_back` method in the `vector` class), it will not be detected by Rex. While the accuracy of read and write detection can never be 100%, future work should aim to improve this relation.

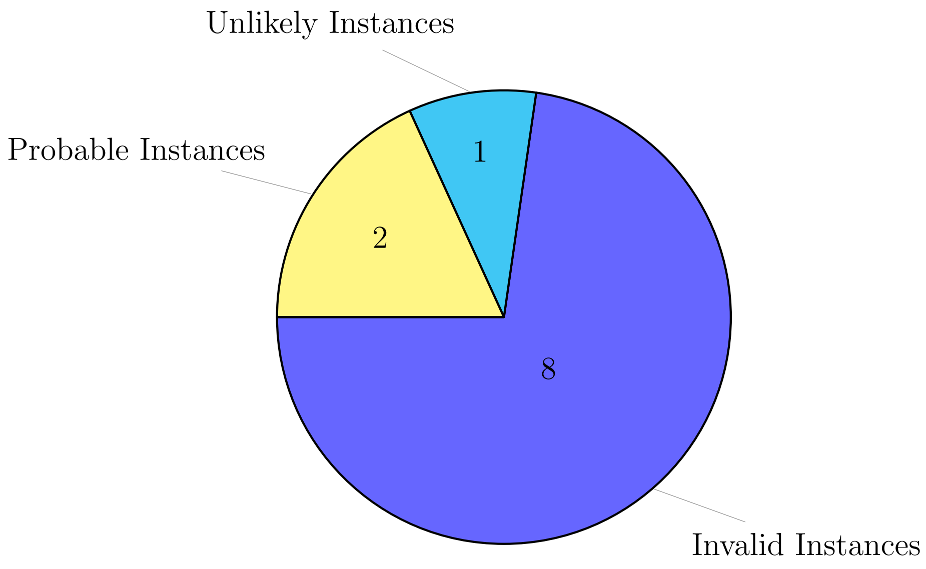


Figure 5.5: The race condition hotspot in Autonomoose broken down by severity.

In addition to completeness, the correctness of the multiple publisher and race condition hotspots were investigated. Since the multiple publisher hotspot detected zero instances where there were multiple topics feeding into a single callback function the correctness of this hotspot cannot be verified. However, the lack of this hotspot's presence in Autonomoose is encouraging since the developers felt the need to ensure each instance of feature communication had a separate communication channel. Having separate channels cuts down on the code complexity and ensures that unexpected feature interactions of this nature do not occur.

Each of the eleven detected race condition hotspot instances were investigated for correctness using manual inspection. For each race condition instance detected, the source code pertaining to that instance was manually inspected. Then, from this inspection, each reported instance was categorized into a severity categories which indicates whether that instance should be more thoroughly inspected. The three severity categories are: *invalid*, *unlikely*, and *probable*. The invalid category denotes instances that do not fit the hotspot. The unlikely category denotes instances that could potentially result in unexpected interactions between features but are unlikely to. Lastly, the probable category is used to denote instances that could result in potential interactions and should be inspected more thoroughly through manual analysis.

Figure 5.5 breaks down the results of classifying each instance. While the majority of reported instances were *invalid*, this was due to a particular set of variables in one feature that received the exact same value from all callback functions that wrote to them. The single unlikely instance is one where multiple callbacks update a time variable to get the most recent received message. This time variable is used to determine when a timeout occurs. This instance is of little concern since a timeout is likely to not occur due to a race condition. Lastly, the 2 probable instances involve multiple callback functions writing 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 case where one state change overrides the other.

5.2.4 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 Grok scripts that detect control flow hotspots, the following results were obtained summarized in Table 5.6. Since these hotspots generate a lot of different information, this table divides the results into four different categories: `Instances`, `Callback Functions`, `Direct`, and `Indirect`. 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. Last, 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].

There are 650 instances of behaviour alteration where a feature has its behaviour altered due to messages received from a ROS topic. While there are a large number of reported instances, these instances only occur in 35 of Autonomoose's 47 callback functions. Further,

[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 Name	**Instances**	**Callback Functions**	**Direct**	**Indirect**
Behaviour Alteration	650	35	11	16
Publish Alteration	64	22	7	15

Table 5.6: Result of detecting control flow hotspots in Autonomoose.

these callback functions are only present in 8 of 14 Autonomoose features. For the features reported in this hotspot, four of them are essential in coordinating the vehicle's route. As such, if one of these features receives a message, it is likely that it will change state and alter the vehicle's route.

For the publish alteration hotspot, there are 64 cases where a feature publishes data due to ROS messages it has received. These 64 instances are spread across 22 callback functions. It was expected that this result would be lower than the behaviour alteration hotspot since 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 to other features.

Verification

To detect this hotspot, the following information is required: (i) `publishers`, (ii) `subscribers`, (iii) `topics`, (iv) `functions`, (v) `classes`, (vi) `publish` messages, (vii) `subscribe` messages, (viii) function `calls` (ix) variable writes, (x) **variables writing to other variables**, (xi) **variables affecting whether a function call is made**, (xii) **variables affecting whether a publish call is made**, (xiii) hierarchies between `classes` and `functions`, and (xiv) **variables that alter control flow**. Information shown in bold has not yet been checked for completeness in previous sections.

The `varWrite`, `varInfluence`, `varFuncInf` relations and the `isControlFlow` needed to be checked for completeness. To do this, a subset of entities in the Rex-generated TA model were compared against the source code to confirm that the TA model accurately represents the source code.

For the `VarWrite` relation, 100 variables were randomly selected in Autonomoose. Then, the source code was manually scanned to find each of these variables and it was recorded whenever any of those variables received data from another variable. The results of this analysis were compared to the Rex-generated TA model to determine the accuracy of `VarWrite`. Out of these 100 random variables that were tracked, the Rex model was

Element Type	Source Code	TA Total	Correct	Incorrect	Accuracy
`VarWrite`	100	100	97	3	97.00%
`VarInfluence`	52	52	51	1	98.08%
`VarFuncInf`	100	100	93	7	93.00%
`isControlFlow`	164	164	154	10	93.90%

Table 5.7: Completeness of ROS-based entities required to detect control flow hotspots.

found to be correct 97% of the time. There were 3 false negatives that occurred since Rex ignores situations where variables are parameters in functions. For example, an expression like $var = function(varA, varB)$ is not visited despite $varA$ and $varB$ *technically* writing to var. Further, while the accuracy of this relation is good, this relation does not track the flow of data between function calls. Therefore, the model will not contain any variable writes between a variable used in a function call and its associated parameter. Future work should investigate developing a relation that can track this type of dataflow.

For the `VarInfluence` relation, since there are only 52 publishers, all publisher objects and the associated `publish` calls were inspected. For each `publish` call, manual analysis was conducted to note whether it was nested under a control structure and which variables affect the decision portion 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, the Rex-generated model is correct for 51 of them. This means that it accurately identifies all the publishers under a control structure and the variables that indirectly affect each. There was one false negative case where the Autonomoose model was incorrect.

For the `VarFuncInf` relation, 100 functions from across the Autonomoose project were randomly selected for manual analysis. Like the `VarInfluence` relation, for each function, it was noted whether there was a call to that function nested under a control structure and, if so, the variables that participate in the decision portion of that control structure. This was then compared to the `VarFuncInf` relation in the Autonomoose model. It was found that, 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; the model marks them as incorrectly being under a control structure.

For the `isControlFlow` attribute, 167 variables were inspected across ten different Autonomoose components. Each variable was manually inspected to determine whether it affects the decision condition of a `for`, `switch`, `if`, or `while` statement. The results from this manual analysis was then compared to the Rex-generated model. Overall, the `isControlFlow` attribute is correct 92.22% of the time. Of the ten incorrect cases, Rex predicted them all to participate in a control structure making them false positives.

Determining the correctness of these two hotspots is important. To do this, each reported instance for the two hotspots were manually verified and then classified into three categories: *invalid*, *unlikely*, and *probable*. As before, the invalid category denotes instances that do not properly fit into the hotspot definition. 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. Last, the probable category is for instances where the overall behaviour of the feature is impacted and where multiple features are involved. These instances should be investigated further.

First, for behaviour alteration, Figure 5.6 breaks down the results from classifying each instance. Overall, there are 505 invalid instances, 89 unlikely instances, and 56 probable instances. While the number of invalid instances is high, the vast majority 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 section of control structures. As such, if any callback function uses these macros, this hotspot will report those macros as instances of behaviour alteration. Since these macros just responsible for logging data to console and altering the logging frequency, they can be safely discarded and classified as invalid instances. Future versions of Rex should attempt to detect these cases and ignore them to avoid overwhelming the user.

For unlikely instances, while they are not as strong as probable instances, they do technically affect a node's behaviour. In these 89 instances, values from a variable go on to slightly affect how the feature performs. For instance, a large number of these cases are variables that participate in the terminating condition of a `for` loop. Other cases included in this severity level include instances where 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 be ignored.

For an instance to be considered a probable instance, it has to write to a variable that goes on to modify the state of the node. In many of these cases, the variables modified are fields inside the node that keep track state across the callback functions and alter the type of computations performed. It is recommended that each of these 56 instances should be investigated further.

For the publish alteration hotspot, Figure 5.7 breaks down the results from classifying each instance. Overall, of the 64 instances reported, 12 are invalid, 27 are unlikely, and 25 are probable. Of the 12 invalid instances, the majority are merely used for debugging so developers can see real-time information about the vehicle. The publish call involved in each of these instances publishes to a debugging topic.

For the 27 unlikely instances, there is one overall pattern that describes these instances.

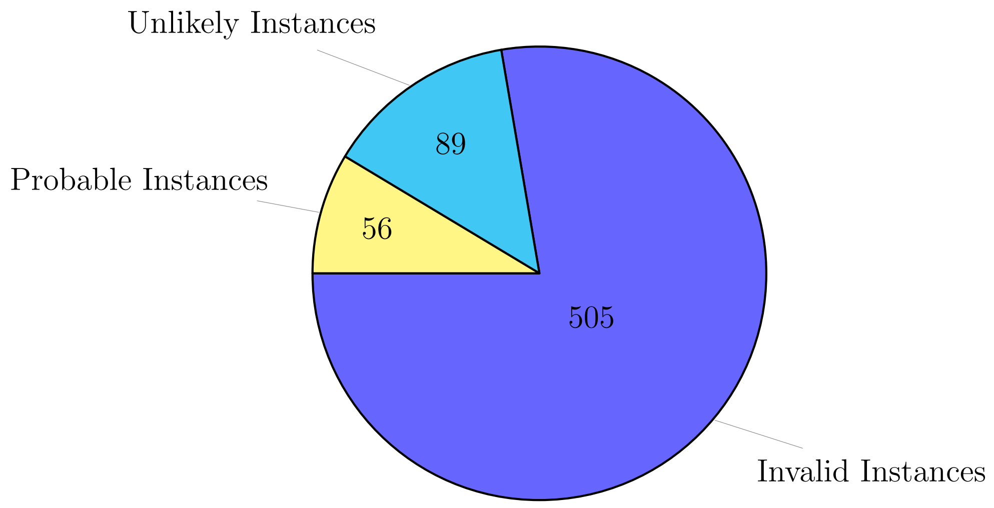


Figure 5.6: The behaviour alteration hotspot in Autonomoose broken down by severity.

Primarily, they have no direct or indirect features that invoke this instance. In other words, the callback function in this instance does not have any features that message it. Instead, many of these instances receive information from the car such as braking or gear information, process it, and pass it on. These areas are unlikely to be problematic since they are at the top of the Autonomoose stack and are simply responsible for forwarding data based on vehicle messages.

Lastly, for the 25 probable instances, the publish calls involved in these instances tend to pass "important" messages to recipient features. This could include route information or information about vehicle state. Further, these instances not only affect the publishing behaviour of the feature but also cause the node to change state. Further investigating these reported instances using manual analysis is important as they are critical to how Autonomoose operates.

5.2.5 Verification with Autonomoose Developers

In the case study, each reported instance for the race condition, behaviour alteration, and publish alteration hotspots were classified into three severity groups: *invalid*, *unlikely*, and *probable*. Since I manually classified these instances, I verified these instances and classifications with developers from the Autonomoose project. For each of these three hotspots, a subset of the reported instances were presented to Dr. Michal Antkiewicz, the

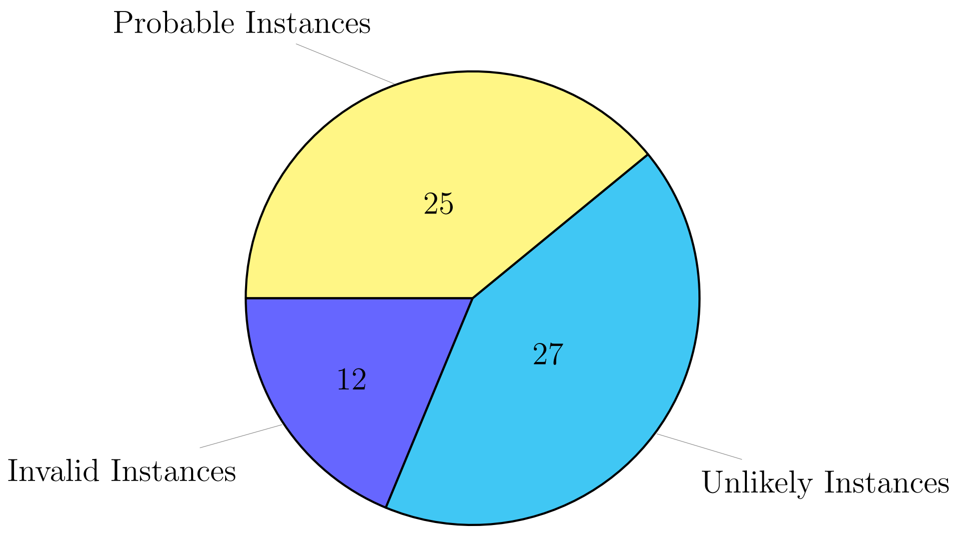


Figure 5.7: The publish alteration hotspot in Autonomoose broken down by severity.

lead research engineer on the Autonomoose project. For each instance, Dr. Antkiewicz would determine whether it was valid and would classify it as either *unlikely* or *probable*.

For each of the three hotspots, all invalid instances were discarded[3] and then ten random instances were selected for each hotspot type. Since the race condition hotspot had only three non-invalid instances, all three were presented.

Race Condition

For the race condition hotspot, all three non-invalid instances were shown to Dr. Antkiewicz. As per Section 5.2.3, I manually classified two as probable and one as unlikely. Each of these three instances were shown to Dr. Antkiewicz and he characterized two as probable and one as unlikely. These classifications matched the ones that I made.

For the two probable instances, Dr. Antkiewicz 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 subject those callback functions to more thorough analysis.

[3]Invalid instances are cases that do not fit the hotspot definition as they are errors that result from invalid entries in the Autonomoose TA model. As such, these are not useful to present to Dr. Antkiewicz.

Behaviour Alteration

For the behaviour alteration hotspot, ten random instances were chosen amongst the 145 non-invalid instances. As per, Section 5.2.4, I manually classified six as unlikely and four as probable. Interestingly, Dr. Antkiewicz found that all the instances presented were not really useful. Based on the instances presented to him, he determined that this hotspot was not overly useful.

One reason that might explain why Dr. Antkiewicz did not find this hotspot to be useful in determining code fragments to inspect further is that only ten random instances were selected out of 145. Such a small sample might mean that if there were some useful instances, they likely were not have been shown to Dr. Antkiewicz. This is concerning since having this much noise in the results might mean that developers could miss an important hotspot instance. Future work should investigate developing sensitivity levels for each hotspot to attempt to cut down on the number of results. Further, 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 Dr. Antkiewicz.

Publish Alteration

For the publish alteration hotspot, ten random instances were chosen amongst the 53 non-invalid instances. Of these I manually classified five of these as unlikely and five as probable. After showing each instance to Dr. Antkiewicz, he categorized six as probable and four as unlikely. Of his classifications, his matched with mine for 60% of the instances.

The reason there was a difference in how Dr. Antkiewicz and I classified these instances is due to what each of us deemed as important. Dr. Antkiewicz was interested in all 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 "known" 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 important to inspect further. Many of these instances with complex traces would be difficult for a developer to detect without the use of tools.

Chapter 6

Conclusions

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

6.1 Contributions

The contributions presented in this thesis are as follows:

- The development of the *ClangEx* fact extractor which is capable of extracting general information from C and C++ source code. ClangEx was written as a general purpose extractor with the intention for it 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 from C and C++ source code. 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 seven different 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 seven hotspots are: *component-based communication*, *dataflow communication*, *loop detection*, *multiple publishers*, *race conditions*, *behaviour alteration*, and *publish alteration*.

- A case study that demonstrates the feasibility of detecting these hotspots in automotive software system. The case study was conducted on the Autonomoose autonomous driving platform which contains over 20,000 lines of code in the main project and 7.3 million lines of code of libraries [66]. Each hotspot type was run on the Autonomoose TA model and the instances were classified into severity levels. These severity levels were verified by Dr. Michal Antkiewicz of the Autonomoose project.

6.2 Limitations

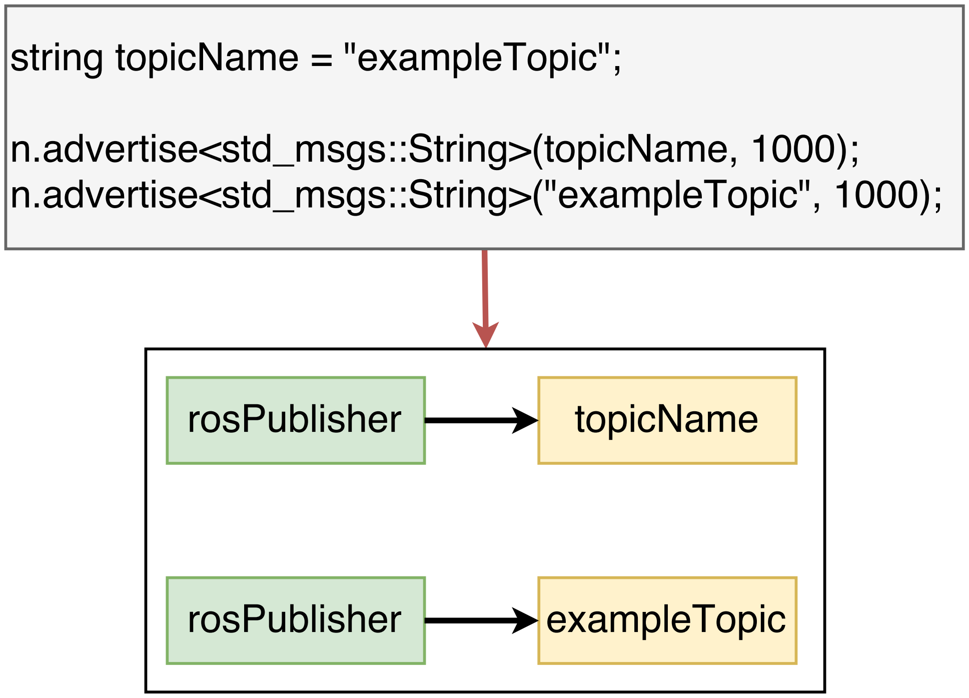
While the relational algebra toolchain is effective at detecting feature interaction hotspots in distributed, message-passing systems, there are several limitations to this methodology. The majority of these limitations are due to limitations of static analysis.

First, the relational algebra toolchain is a static analysis approach. While fact extractors are capable of modeling the entire codebase of a project, these extractors share limitations that plauge other static analysis tools. Since static analysis requires the examination of the code without execution, some code behaviours such as threads or message-passing cannot be modelled accurately. This leads tools to reason about the system in an approximate manner [67]. Interactions shown in models developed using fact extractors might not actually be present when the software is running. Due to these drawbacks, detecting hotspots using this approach is not a fully automated process. Since there is a possibility of false positives and negatives, this toolchain is merely a technique that aims to narrow down the amount of manual analysis 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. While relying on these conventions was sufficient when analyzing Autonomoose, Rex may not be as successful in generating a correct model for other ROS-based projects. An example of this is that, when creating publisher and subscriber objects in a ROS-based component, Rex assumes that the topic name passed to the publisher and subscriber object is a string literal. While programmers are able to pass string variables to the publisher or subscriber objects constructor instead, Rex will use the name of the variable passed as the topic name[1]. Figure 6.1 shows an example of this limitation. In it, while the two advertise

[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.

functions create publishers which publish data to the same topic, Rex sees these two topics as different.



```
string topicName = "exampleTopic";

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

Figure 6.1: C++ code that would cause Rex to generate an incorrect model.

Lastly, there are limitations with how the `read`, `write`, and `varWrites` relations in Rex and ClangEx operate; all of these relations are responsible for tracking variable usage in expressions. In C and C++, detecting variable reads and writes is not as straightforward as in other languages such as Java[2]. Due to aliasing, developing a precise variable read and write detector is impossible in static analysis since variable values can be used as addresses or values. If it were possible to statically determine variable values, compilers could simply record the output of a program rather than having to compile it[3]. 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 a variable uses the indirection or address operator it will be treated the same as variables not using

[2]Eclipse for Java has a feature that statically allows a user to view all variable references directly inside the IDE.

[3]See this StackOverflow discussion:
https://stackoverflow.com/questions/15252137/monitoring-variable-accesses-in-c-c

these operators. As shown in Figure 6.2, this can cause a possible analysis error.

```
int main(){
        //Variables being created.
        int varOne = 10;
        int* varTwo = 20;

        //Variable being written to.
        varOne = 5;

        //Is this variable being read, written to, or neither?
        varTwo = &varOne;

        return 0;
}
```

Figure 6.2: An example of why determining variable accesses is flawed.

6.3 Future Work

There is much work that needs to be conducted to further the contributions presented in this thesis. These areas of future work fall into four main categories: (i) 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; (ii) updating ClangEx and Rex to extract more information from C and C++ source code and to improve current detection; (iii) exploring whether analyzing additional software artifacts can improve hotspot detection quality; and (iv) developing more hotspots and creating a method to automatically classify the severity of each hotspot instance. Each of these areas of future areas of work will be further explained in this section.

First, while this thesis examines a distributed automotive system where features are separated into different modules and communicate using a common framework, 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. While Section 5.1 argues that analyzing Autonomoose is sufficient due to the ROS achitecture's similarity to the CAN bus architecture, a lot of work is still required to determine how feature interaction information can be extracted from CAN bus systems. Further, the presence of the AUTOSAR framework means that extractors targeting traditional automotive systems must be able to understand the AUTOSAR API to detect feature communications. Once this has been achieved, it is important to conduct a case study using an AUTOSAR and CAN-based automotive system to ensure the relational algebra toolchain can be used and to determine whether the hotspots postulated in this thesis are still applicable. Although AUTOSAR is not currently used in many traditional automotive systems, most automotive companies express their intention to move to the AUTOSAR in the future [68]. The advantage of focusing on AUTOSAR in future work is that this framework divides the software system into multiple layers each abstracted from layers 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. While 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[4]. These could allow users to track which variables are influenced by timers or services. Further, while most of the relations in Rex were verified to have over 90% accuracy, it would be good to further improve their accuracies to produce better models.

While the relational algebra toolchain is effective at processing C and C++ code, it is also capable of extracting information from other *structured* artifacts. As an area of future work, it would be interesting to develop extractors and tools that can process additional project resources including build scripts. 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

[4]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.

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.

Lastly, 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. While the hotspots presented in this thesis tend to be generic, some platform-specific hotspots could be introduced. For instance, ROS has the concept of timers which call specific functions at a particular frequency; developing hotspots that involve those might be useful. Users could discover when a feature publishes data as a result of a timer. Additionally, it might be beneficial to develop a filtering method for hotspot results to prevent users from being overwhelmed. For instance, as per Chapter 5, running the behaviour alteration hotspot on Autonomoose returned over 650 results and many of these results were deemed not useful by Autonomoose research engineers. By being inundated by so many results, there is a possibility that developers might miss some important instances. Filtering out results might involve examining each instance to predefined patterns that dictate "severity" or look at the trace length of an instance. For instance, for the publish alteration hotspot, an instance where a callback function immediately publishes data might be less severe 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] J. Cohen, "11 proven practices for more effective, efficient peer code review," jan 2011.

[22] 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.

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

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

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

[26] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphvizopen source graph drawing tools," in *International Symposium on Graph Drawing*, pp. 483–484, Springer, 2001.

[27] S. Jänicke, A. Geßner, G. Franzini, M. Terras, S. Mahony, and G. Scheuermann, "Traviz: a visualization for variant graphs," *Digital Scholarship in the Humanities*, vol. 30, no. suppl_1, pp. i83–i99, 2015.

[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] K. R. Jayaram and P. Eugster, "Program analysis for event-based distributed systems," in *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, (New York, NY, USA), pp. 113–124, ACM, 2011.

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

[32] SciTools, "Scitools' understand," 2017.

[33] SciTools, "Writing codecheck scripts," 2017.

[34] 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.

[35] A. Hadoop, "Hadoop," 2009.

[36] C. Boost, "Libraries," 2012.

[37] S. A. G. (SWAG), "Javex - java fact extractor," apr 2010.

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

[39] S. A. G. (SWAG), "Ldx and bfx pipeline," 2010.

[40] R. Holt, "The tuple-attribute (ta) language," 1997.

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

[42] J. Uhl, "Rigi standard format," 1996.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

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

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

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

[51] 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.

[52] C. O.-S. Compiler, "Language compatibility."

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

[54] E. Ruelas, "Symptoms of a bad or failing electronic brake control module (ebcm)," Jan 2016.

[55] S. Corrigan, "Introduction to the controller area network (can)," *Texas Instrument, Application Report*, 2008.

[56] I. Standard, "Iso 11898, 1993," *Road vehicles–interchange of digital information–Controller Area Network (CAN) for high-speed communication*, 1993.

[57] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces.," in *USENIX Security Symposium*, San Francisco, 2011.

[58] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar–a worldwide standard is on the road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, 2009.

[59] J. Alexandersson and O. Nordin, "Implementation of can communication stack in autosar," 2015.

[60] 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.

[61] 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.

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

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

[64] Bosch Semiconductors and Sensors, *CAN Specification*, 2.0 ed., sep 1991.

[65] J. M. O'Kane, *A Gentle Introduction to ROS*, vol. 2.1.3. University of South Carolina, apr 2014.

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

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

[68] 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 the bfx64 Extractor

bfx64 is a fact extractor designed to collect basic program information from object files. Based upon Jingwei Wu's *BFX* extractor [45], 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. While this fact extractor was not covered in this thesis, bfx64 was developed during my research at the University of Waterloo.

A.1 Installing bfx64

Before bfx64 can be installed, there are several libraries and tools that must be present on the target system. First, since 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 A.1.1 describes CMake, Boost, and ELFIO installation. Section A.1.2 describes how to build bfx64.

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 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. Since 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
```

A.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 bfx64 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.

A.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 A.2.1 highlights basic commands and Section A.2.2 highlights advanced commands. For more information about a command, refer to bfx64's help screen by running `bfx64 --help`.

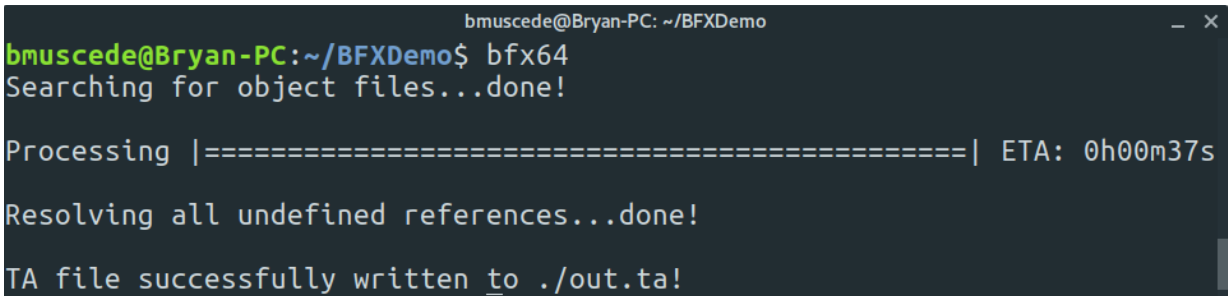
A.2.1 Basic Usage

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

A.2.2 Advanced Features

There are also some advanced arguments that provide more fine-grained control over bfx64. Table A.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 A.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, while that user wants to process most of the object files inside `bfx64Demo`, they do not want to process the

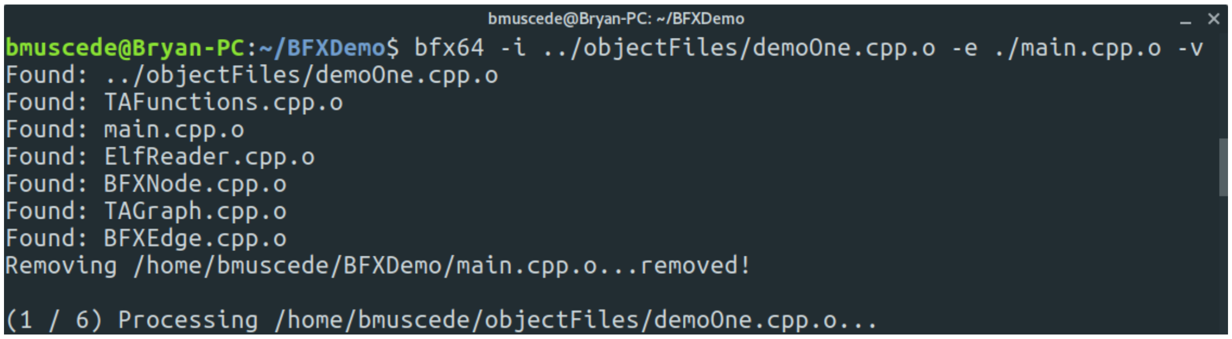

```
bmuscede@Bryan-PC: ~/BFXDemo
bmuscede@Bryan-PC:~/BFXDemo$ bfx64 -i ../objectFiles/demoOne.cpp.o -e ./main.cpp.o -v
Found: ../objectFiles/demoOne.cpp.o
Found: TAFunctions.cpp.o
Found: main.cpp.o
Found: ElfReader.cpp.o
Found: BFXNode.cpp.o
Found: TAGraph.cpp.o
Found: BFXEdge.cpp.o
Removing /home/bmuscede/BFXDemo/main.cpp.o...removed!

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

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

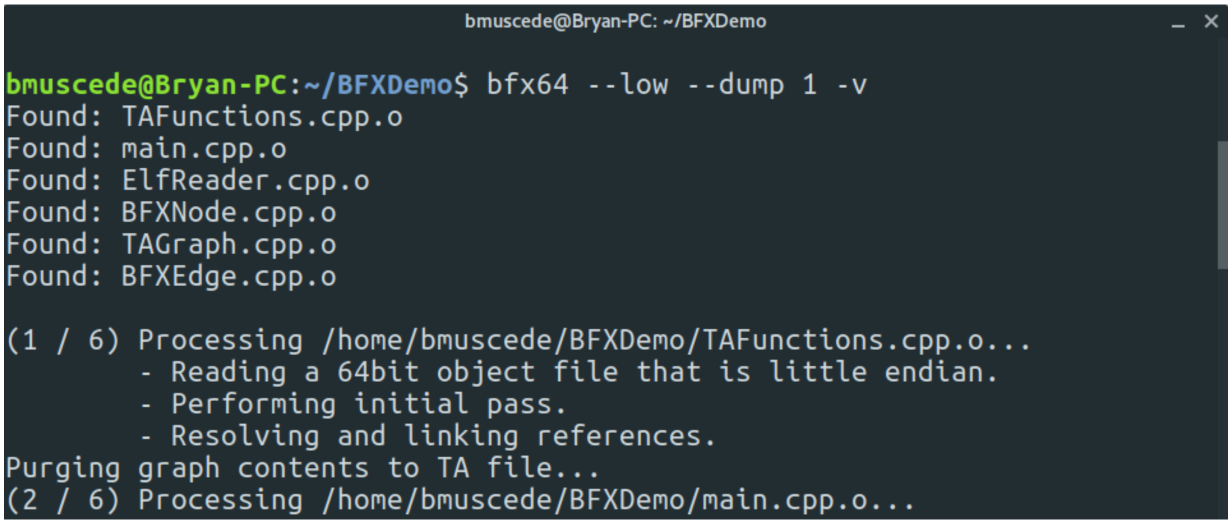
`main.cpp.o` object file. Figure A.2 shows how this can be achieved using bfx64. In this example the `-i` argument is used to tell bfx64 to include the `demoOne.cpp.o` object file while the `-e` argument is used to tell bfx64 to exclude the `main.cpp.o` object file. Lastly, the `-v` flag puts bfx64 into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

Low-Memory Arguments

Since 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. Since 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 A.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. Since 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

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

Appendix B

Installing & Using ClangEx/Rex

B.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**, and **Clang** 5.0 or greater. CMake is used to build ClangEx and Rex, Boost is a collection of C++ libraries that 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. Section B.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 B.1.2 describes how to build ClangEx from source and Section B.1.3 describes how to build Rex from source.

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 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

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
```

Clang

While 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.

B.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.

B.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 Rex 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.

B.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 B.1 shows the steps required to process C or

C++ projects using ClangEx. In this figure, green boxes represent mandatory steps while 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.

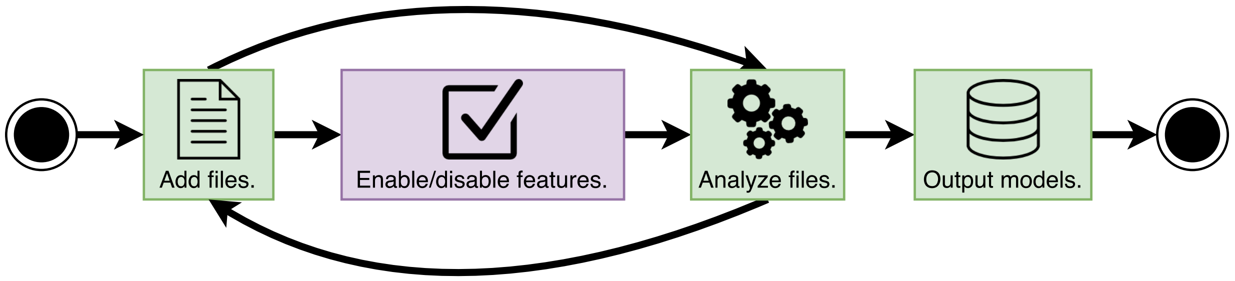


Figure B.1: The ClangEx processing pipeline.

When ClangEx is first started, it displays an empty prompt and waits for a user's commands. Table B.2 shows a list of commands that ClangEx accepts. 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 B.1.

B.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. While C and C++ files can technically end with any extension, this search tool only looks for files that end with the following extensions: (i) .c; (ii) .cc; and (iii) .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.

B.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 B.1 shows all the language features that can be enabled or disabled.

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`.

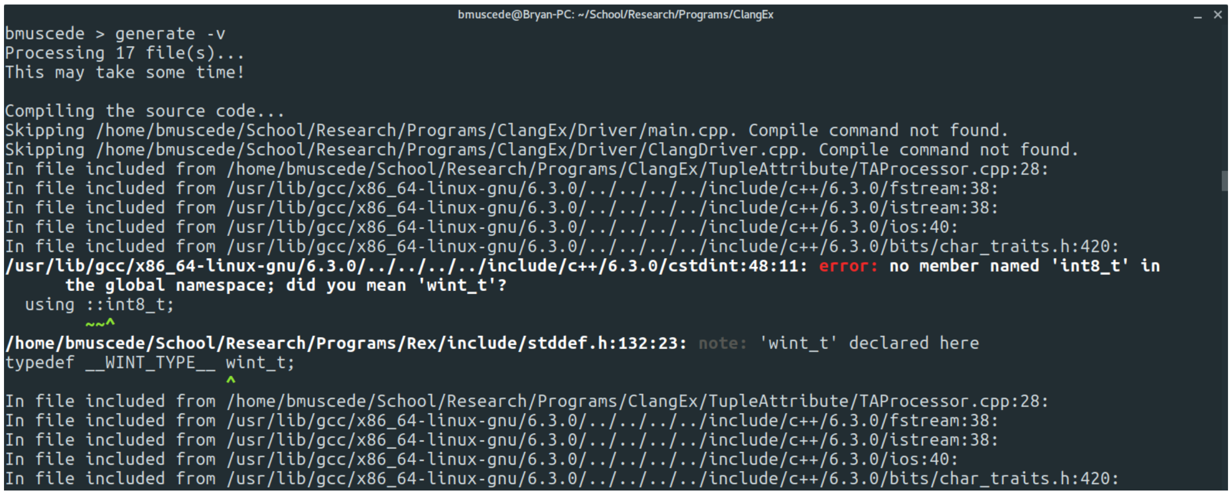
B.2.3 Step 3 - Analyzing Files

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

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 B.1: ClangEx language feature names.

analysis is `generate`. Figure B.2 shows ClangEx while it is analyzing source code. Note that ClangEx will output any compiler errors or warnings that are encountered during analysis.



```
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 B.2: ClangEx while running on some unspecified source code.

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 *blob* 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 blob 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 blob 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).

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.

B.2.4 Step 4 - Outputting Models

Once ClangEx has generated models, they need to be outputted to disk as TA files. Since 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.

B.3 Using Rex

Since 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 B.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: (i) adding files to the processing queue; (ii) building a model of all source files in the processing queue; (iii) resolving which classes belong to which features; and (iv) outputting all generated models to disk as tuple-attribute files.

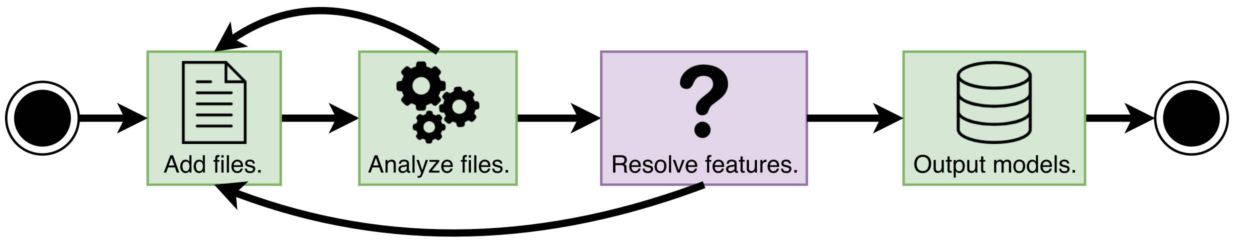


Figure B.3: The Rex processing pipeline.

Since the Rex pipeline is almost identical to the ClangEx pipeline, it uses the same commands as ClangEx. Table B.3 gives common Rex commands that can be used while running the extractor. 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.

B.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.

B.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. While 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 blob 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 B.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 B.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 tells ClangEx to process the files using the *blob* methodology.

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 B.2: 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 B.3: An overview of common Rex commands.

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 Component-Based Communication

Figure C.1 shows the Grok script that detects component-based communications. 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. Lines 14 and 15 are the central part of the script; line 14 joins the `publish` and `subscribe` relations together using the selection operator and then lifts this joined relation to the parent classes. Line 15 gets any indirect communications by taking the lifted `direct` relation and then gets the transitive closure (the `+` operator). Line 15 also uses set difference to remove any direct entries from that list. Finally, lines 17 through 22 print the relations. The reason `inv`

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

`@label o <RELATION_NAME> o @label` is needed in lines 19 and 22 is because without it, the IDs of the features communicating would be printed rather than their human-readable name.

```
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 //Performs lifting and gets direct/indirect calls.
14 direct = contain o (publish o subscribe) o (inv contain);
15 indirect = (direct+) - direct;
16
17 //Prints the results.
18 print "Direct Messages:"
19 inv @label o (compContain o direct o inv compContain) o @label;
20
21 print "Indirect Messages:";
22 inv @label o (compContain o indirect o inv compContain) o @label;
```

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

C.1.2 Dataflow Communication

Figure C.2 shows the Grok script that detects instances of the dataflow communication hotspot. For brevity, while also required in this script, lines 1 through 11 of the component-based communication script in Figure C.1 were omitted from this script.

Like in the component-based communication script, line 2 of Figure C.2 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 5 through 18 gets the indirect instances of communication. To do this, the publish and subscribe relations are joined again 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

```
1  //Gets the direct communications
2  direct = contain o (publish o subscribe) o (inv contain);
3
4  //Combines publishers and subscribers with function calls.
5  rosComm = publish o subscribe;
6  fullCall = rosComm + call;
7  fullCall = fullCall+;
8
9  //Gets a list of publishers and subscribers.
10 publishSet = $INSTANCE . {"rosPublisher"};
11 subscribeSet = $INSTANCE . {"rosSubscriber"};
12
13 //Ensures publishers start and subscribers end.
14 comms = publishSet o fullCall o subscribeSet;
15
16 //Gets the indirect communication.
17 indirect = contain o comms o inv contain;
18 indirect = indirect - direct
19
20 //Prints the results.
21 print "Direct Messages:";
22 inv @label o (compContain o direct o inv compContain) o @label;
23
24 print "Indirect Messages:";
25 inv @label o (compContain o indirect o inv compContain) o @label;
```

Figure C.2: Grok script that detects the dataflow communication hotspot.

of communications between features in the project. Line 7 gets the transitive closure of the `rosComm` relation. With the transitive closure of `fullCall` generated, lines 10 and 11 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 14, the script filters out results from `rosComm` that do not start with a publisher and end with a subscriber. This gives us all the dataflow communications between features. Last, line 17 lifts the indirect relation to the class level and line 18 removes and direct results. The resulting values are then printed in lines 22 through 25.

C.1.3 Loop Detection

Figure C.3 shows the Grok script that detects any loops in the feature communication graph. Again, for brevity, lines 1 through 11 of the 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 two scripts 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.

```
 1 //Generates a list of classes pointing back to themselves.
 2 classes = $INSTANCE . {"cClass"}
 3 classes = (classes o contain) o inv contain;
 4 
 5 //Generates the call graph.
 6 fullCall = (publish o subscribe) + call;
 7 fullCall = fullCall+;
 8 
 9 //Generates the direct loops.
10 classComm = contain o (publish o subscribe) o (inv contain);
11 direct = classComm ^ classes;
12 
13 //Get the component-based indirect results.
14 indirectComp = ((classComm+) ^ classes) - direct;
15 
16 //Gets the dataflow indirect results.
17 dataflowComm = contain o publish o fullCall o subscribe o inv contain;
18 indirectDataflow = dataflowComm ^ classes;
19 indirectDataflow = indirectDataflow - direct;
20 
21 //Prints the results.
22 print "Direct Loops:";
23 inv @label o (compContain o direct o inv compContain) o @label;
24 
25 print "Indirect Component Loops:";
26 inv @label o (compContain o indirectComp o inv compContain) o @label;
27 
28 print "Indirect Dataflow Loops:"
29 inv @label o (compContain o indirectDataflow o inv compContain) o @label;
```

Figure C.3: Grok script that detects the loop hotspot.

Lines 2 and 3 in Figure C.3 generate a nonsense relation 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. While it may appear useless, this `classes` relation is used later to determine whether there is a direct or indirect loop. Like in lines 5 through 7 of Figure C.2, lines 6 and 7 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 10 and 11 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. Since this gets all communications and not just loops, to remove any non-loops, the intersection operator (ˆ) is used on the `classComm` relation and `classes` relation. This means only entries in the `classComm` relation that has the same domain and range are kept. These are the direct loops.

Line 14 generates a list of indirect component-based communication loops. This is achieved by computing the transitive closure of classes that communicate, only keeping the loops, and then removing any of the direct loops from the results. Lines 17 through 19 get the indirect dataflow loops. This is achieved by taking the `fullCall` set (created in lines 6 and 7), ensuring that it only starts with a publisher and ends with a subscriber, and then removing any non-loops. Lastly, lines 22 through 29 print the results of each.

C.2 Multiple Publisher Hotspots

C.2.1 Multiple Input

Figure C.4 shows the Grok script that detects the multiple input hotspot. For brevity, lines 1 through 11 of the 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` 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, since this script is only interested in

```
 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 //Purges communications with one instance.
 9 for entry in dom inset {
10   curNum = {entry} . inset;
11   if #(curNum - {"1"}) == 0 {
12     inset = inset - ({entry} X {"1"});
13   }
14 }
15 
16 //Prints the results.
17 print "Subscribers that have Multiple Component Communications:"
18 inv @label o inset;
19 
20 //Now, dives deeper.
21 for item in dom(inset) {
22   cbFunc = {item} . call;
23   cFunction = (rng(cbFunc o @label));
24   cClass = rng((dom(contain o rng{item})) o @label);
25 
26   //Print the items.
27   print "Callback Function " + cFunction + " (" + cClass + ") - ";
28 
29   //Get the publishers.
30   inv @label o (compContain . (direct . {item}));
31 }
```

Figure C.4: Grok script that detects the multiple publishers hotspot.

subscribers that receive messages from numerous features, this script filters out any entries in `inset` relation where the number of occurrences is only 1. To do this, lines 9 through 14 loop through the `inset` relation on the domain, and check if that subscriber only has one feature that messages it. 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 17 and 18 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 22 through 24 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 24 involves taking the `contains` relation and joining the subscriber to the range of that relation. This gets the parent class. Line 27 prints the function and class the subscriber is in and then line 30 prints all the features that message that subscriber.

C.2.2 Race Condition

Figure C.5 shows the Grok script that is used to detect the race condition hotspot. While 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 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. Since 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>`.

Lines 6 through 21 contain a `for` loop block that loops through every variable in the `vars` relation created in the previous line. Since 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 7 takes the current variable and gets all instances in the `vars` relation where that variable is written to. By doing this, the script can then check if that variable is written to by more than one callback function. Lines 10 through 20 only process variables where more than one callback function writes to them. If a variable is written to by more than one callback function, the name of the variable is printed in line 12. Then line 15 gets all callback functions that modify this variable and prints each to the screen.

```
//Get a list of callback functions.
callbackFunc = rng (subscribe o call);

//Determines all the variables that are modified by each callback function.
vars = callbackFunc o write;
for curVar in rng vars {
  specific = vars . {curVar};

  //Gets the number of instances of that variable.
  if (#specific > 1) {
    //Deal with cases where multiple callbacks modify.
    print "For the " + curVar + " variable:";

    //Gets the callback functions that push to that variable.
    callbacks = dom specific;
    for cb in callbacks {
      //Prints the callback.
      {cb} . @label;
    }
  }
}
```

Figure C.5: Grok script that detects the race condition hotspot.

C.3 Control Flow Hotspots

C.3.1 Behaviour Alteration

Figure C.6 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 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 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

```
 1 //Gets the direct and indirect relations.
 2 direct = contain o publish o subscribe o call;
 3 indirect = contain o publish o subscribe o inv contain;
 4 indirect = indirect+;
 5 
 6 //Generates relations to track the flow of data.
 7 callbackFuncs = rng(subscribe o call);
 8 controlFlowVars = @isControlFlow . {"\"1\""};
 9 masterRel = varWrite + call + write;
10 masterRel = masterRel+;
11 
12 //Gets the behaviour alterations.
13 behAlter = callbackFuncs o masterRel o controlFlowVars;
14 print "There are " + #behAlter + " cases of behaviour alteration.";
15 print "Across " + #(dom behAlter) + " callback functions.";
16 
17 //Loops through each callback function.
18 for item in dom behAlter {
19   //Gets the callback function name.
20   {item} . @label;
21   print "";
22 
23   //Print the variables it affects.
24   print "Affects Variables:"
25   inv @label . ({item} . behAlter);
26   print "";
27 
28   //Print features that directly influence.
29   print "Influenced By - Direct:"
30   dirInf = direct . {item};
31   inv @label . dirInf;
32 
33   //Print features that indirectly influence.
34   print "Influenced By - Indirect:";
35   inInf = (indirect . (direct . {item})) - dirInf;
36   inv @label . inInf;
37   print "";
38 }
```

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

`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`, and `write` 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.

Since 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.7 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 component-based communication script in Figure C.1. Further, since this script is a slight modification of the behaviour alteration script, there are some similarities in how this hotspot is detected.

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

```
//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.7: Grok script that detects the publish alteration hotspot.

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.

Since 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.