Improving the Precision of Analyses Queries in Factbase Models of Software Systems

by

Xiang Yun Ke

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

Waterloo, Ontario, Canada, 2024

© Xiang Yun Ke 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large software systems are developed by multiple teams of software engineers, each working on different components that are supposed to work together. Each component is responsible for a subset of system functionality and the components communicate with each other to react to information received from hardware sensors and user inputs. It is often infeasible to perform manual code reviews on large software systems because the code base may be too large, the components may be written in different languages or language variants, or the concurrency of components can lead to a state explosion of the system's analysis space. To mitigate these challenges, we create a software model consisting of facts about the software system and perform analyses on the model.

Analyses performed on these software models are not always sound and complete. One of the reasons is that the order of execution of facts in the model is unknown, leading to many false-positive results that refer to infeasible execution paths. Our work addresses this problem by extending a fact-based software model with control-flow-graph facts and associating existing facts with their corresponding control flow blocks. Then, the analyses are revised to check that results correspond to execution paths (in terms of control-flow-graph facts) before reporting results to the engineers.

This extra execution-path check causes the revised analyses to exhibit significant performance overhead. To reduce the overall execution time of the analyses, we (1) stage analysis queries so that they work on a subset of the facts to generate partial results incrementally and (2) employ an on-the-fly execution path check that eliminates invalid sub-results within the analysis engine.

Our work is evaluated with ten different analyses performed on six software systems that use the ROS (Robot Operating System) framework for cross-component communication. A detailed precision and performance evaluation was performed on Autonomoose and WISE-ADS, two ROS-based autonomous driving systems. In addition, this thesis adapts our approach to non-ROS systems (in which components communicate via function calls instead of passed messages) and we evaluate that work by analyzing a non-ROS software controller. The controller experiment is designed to test the scalability of our work when applied to large real-world applications.

Acknowledgements

I would like to thank all the people who made this thesis possible.

Firstly, I would like to thank my supervisor, Dr. Joanne Atlee, for guiding me throughout my research. I appreciate all her constructive feedback and suggestions for my thesis. She has always inspired me throughout this whole journey.

I would also like to thank the WISE Automated Driving System (ADS) Lab for sharing their source code and Docker image. The precision evaluation of this research work would be challenging without these valuable resources. Special thanks to Dr. Michal Antkiewicz for helping me set up the experiment environment for the WISE ADS system.

I am also thankful for being a part of an awesome research group. I enjoyed the different activities that we did as a group and all of the meaningful discussions we had. I would also like to thank Rafael for helping me build the software model for one of the subject systems.

Lastly, I would like to thank Dr. Nancy Day and Dr. Chengnian Sun for agreeing to read my thesis and being on my thesis committee. I am grateful for the time and effort they are willing to spend reviewing my work.

Dedication

This is dedicated to Mom, Dad, Grandpa, and Grandma. Thanks for always being supportive and positive towards my studies and life choices.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures x

List of Tables xiv

1 Introduction 1

1.1 Thesis Contribution 5

1.2 Thesis Organization 5

2 Background and Previous Work 7

2.1 Robot Operating System 7

2.2 Rex Fact Extraction and Factbases 8

2.3 Neo4J Graph Database, Cypher, and APOC 12

2.4 Query Analyses 13

2.5 Control Flow Graph 14

3 Extensions to Rex **17**
3.1 parWrite . . . 17
3.2 cReturn . . . 17
3.3 pubVar and pubTarget . . . 19
3.4 Add File Name to Node Identifier . . . 19
3.5 Variables and Default Topic Names . . . 20
3.6 call Facts . . . 20
3.7 CFG Facts . . . 20

4 Neo4J Queries with CFG-Based Checking on ROS Software **25**
4.1 CFG-Based Checking . . . 26
4.1.1 varWrite . . . 26
4.1.2 call . . . 28
4.1.3 parWrite . . . 29
4.1.4 retWrite . . . 30
4.1.5 write . . . 31
4.1.6 publish . . . 32
4.1.7 pubVar . . . 32
4.1.8 pubTarget . . . 32
4.1.9 subscribe . . . 32
4.1.10 varInfFunc . . . 33
4.1.11 varInfluence . . . 35
4.2 Scalability . . . 35
4.2.1 On-the-Fly Checking . . . 35
4.2.2 Shortest Paths . . . 36
4.2.3 Two-Phased Queries (ROS) . . . 36
4.3 Neo4J Queries and APOC . . . 40

5 Query Analysis Evaluation of ROS-Based Software **45**
5.1 Selection of Subject ROS Systems . . . 45
5.1.1 Autonomous Vehicles . . . 45
5.1.2 Open Source Requirements . . . 46
5.1.3 Statistics of the Subject Systems . . . 48
5.2 Subject Analyses . . . 49
5.3 ROS System Experiments . . . 50
5.3.1 Precision Evaluation . . . 51
5.3.2 Performance Evaluation . . . 54

6 Neo4J Queries with CFG-Based Checking on Non-ROS Software **60**
6.1 Two-Phased Queries (Non-ROS) . . . 60
6.2 Neo4J Queries and APOC . . . 61

7 Query Analysis Evaluation of Non-ROS Software **65**
7.1 Pre-Processing . . . 65
7.2 Performance Analysis . . . 67

8 Sources of Imprecision **69**
8.1 Resolved . . . 69
8.2 Improved . . . 69
8.3 Lingering . . . 71

9 Related Work **72**

10 Conclusion **74**
10.1 Limitations . . . 74
10.2 Future Work . . . 75

References **77**

APPENDICES **81**

A Experiment Setup **82**

B Other CFG Extensions **85**

B.1 call . . . 85
B.2 write . . . 86
B.3 publish . . . 86
B.4 pubVar . . . 88
B.5 varInfFunc . . . 88
B.6 varInfluence . . . 89

C Neo4J Queries of Analyses **92**

C.1 Inter-component Call Graph . . . 92
C.2 Behaviour Alteration Analyses . . . 97
C.3 Publisher Alteration Analyses . . . 103
C.4 Inter-Component-Based Communication . . . 106
C.5 Loop Detection . . . 106
C.6 Multiple Publishers . . . 111
C.7 Race Conditions . . . 111
C.8 Intracomponent Call Graph . . . 114
C.9 Recursion . . . 115
C.10 Variable Triangle Communication . . . 116
C.11 Behaviour Alteration Analyses for Non-ROS Applications . . . 116

List of Figures

1.1 Process of analyzing large software systems. 2
1.2 Small C program 2
1.3 Graphical model of small C code example 3
1.4 Results of dataflow cypher query 4

2.1 Diagram for ROS communication framework 8
2.2 Factbase of Small C program. 9
2.3 Factbase of Small C program. 9
2.4 Dataflow cypher query. 12
2.5 Basic blocks of C program. 15

4.1 Graphical pattern of a subquery involving a *varWrite* fact, and its CFG-based check 27
4.2 Graphical pattern of a subquery involving a *call* fact, and its CFG-based check 28
4.3 Graphical pattern of a subquery involving a *parWrite* fact, and its CFG-based check 29
4.4 Graphical pattern of a subquery involving the special variation of a *parWrite* fact, and its CFG-based check 30
4.5 Graphical pattern of a subquery involving a *retWrite* fact, and its CFG-based check 31
4.6 Graphical pattern of a subquery involving a *varInfFunc* fact, and its CFG-based check 34

4.7 The graphical query pattern for an unoptimized cross-component call-graph analysis (for ROS applications) . . . 38
4.8 The graphical pattern for an incremental cross-component call graph query. The top half of the figure depicts the intracomponent query patterns and the bottom half of the figure depicts the system-wide and inter-component query pattern (for ROS applications) . . . 39
4.9 Prefix query for ROS component call-graph. . . . 42
4.10 Intermediate query for ROS component call graph. . . . 43
4.11 Suffix query for ROS component call graph. . . . 44
4.12 System-wide query for ROS cross-component call graph. . . . 44

6.1 New nodes and relationships for cross-component facts. . . . 62
6.2 The intracomponent and system-wide behaviour-alteration queries for non-ROS applications . . . 64

B.1 Graphical pattern of a subquery involving the variation of a *call* fact, and its CFG-based check . . . 86
B.2 Diagram for CFG extension of *write* fact . . . 87
B.3 Graphical pattern of a subquery involving a *publish* fact, and its CFG-based check . . . 87
B.4 Graphical pattern of a subquery involving a *pubVar* fact, and its CFG-based check . . . 88
B.5 Graphical pattern of a subquery involving the variation of a *varInfFunc* fact, and its CFG-based check . . . 89
B.6 Graphical pattern of a subquery involving a *varInfluence* fact, and its CFG-based check . . . 90
B.7 Graphical pattern of a subquery involving the variation of a *varInfluence* fact, and its CFG-based check . . . 91

C.1 The graphical pattern for an incremental cross-component call graph query. The top half of the figure depicts the intracomponent query patterns and the bottom half of the figure depicts the system-wide and inter-component query pattern (for ROS applications) . . . 93

C.2 Prefix query for ROS component call-graph. 94
C.3 Intermediate query for ROS component call graph. 95
C.4 Suffix query for ROS component call graph. 96
C.5 System-wide query for ROS cross-component call graph. 96
C.6 Patterns for behaviour alteration queries for ROS applications 98
C.7 Prefix query for ROS Behaviour Alteration. 99
C.8 Intermediate query for ROS Behaviour Alteration. 101
C.9 Suffix query for ROS Behaviour Alteration. 102
C.10 Full system query for ROS Behaviour Alteration. 102
C.11 Prefix query for ROS Publisher Alteration. 103
C.12 Intermediate query for ROS Publisher Alteration. 104
C.13 Suffix query for ROS Publisher Alteration. 105
C.14 Full system query for ROS Publisher Alteration. 106
C.15 Patterns for publisher alteration queries for ROS applications 107
C.16 Intermediate query for ROS Inter-Component-Based communication. . . . 108
C.17 Full system query for ROS Inter-Component-Based Communication. 109
C.18 Patterns for intercomponent communication queries for ROS applications . 109
C.19 Intermediate query for ROS Loop Detection. 110
C.20 Full system query for ROS Loop Detection. 110
C.21 Full system query for ROS Multiple Publishers. 111
C.22 Multiple publishers query 111
C.23 Intermediate query for ROS Race Conditions. 112
C.24 Full system query for ROS Race Conditions. 113
C.25 Patterns for race condition queries for ROS applications 113
C.26 Query for Intracomponent Call Graph. 114
C.27 Pattern for intracomponent call graph query for ROS applications 114
C.28 Query for Recursion. 115

C.29 Pattern for recursion query for ROS applications 115
C.30 Query for Variable Triangle Communication. 116
C.31 Pattern for triangle variable communication query for ROS applications . . 117
C.32 Prefix query for Non-ROS Behaviour Alteration. 118
C.33 Intermediate query for Non-ROS Behaviour Alteration. 119
C.34 Suffix query for ROS Behaviour Alteration. 120
C.35 Full system query for Non-ROS Behaviour Alteration. 120

List of Tables

2.1 Types of Entity Facts 10
2.2 Types of Relationship Facts 11
2.3 Precision of Feature Interaction Analyses 14

3.1 Types of *cReturn* Relationship Facts 18
3.2 Summary of CFG Relationship Facts (1) 22
3.3 Summary of CFG Relationship Facts (2) 23
3.4 Summary of CFG Relationship Facts (3) 24

5.1 Open Source ROS Projects Statistics 48
5.2 Number of Facts Extracted From Open Source ROS Projects 49
5.3 Autonomoose 3+ Components 52
5.4 WISE ADS 3+ Components 54
5.5 Performance Evaluation of Query Analysis (1) 56

7.1 The Controller Statistics 66
7.2 Length of CFG-Verified Behaviour Alterations 67
7.3 GM Controller - Performance (ms) 68

A.1 Queries Settings 83

Chapter 1

Introduction

Large software systems can consist of millions of lines of code that are distributed among multiple components based on their core functionalities. Such software systems are typically produced by multiple teams of software engineers, which means it is relatively difficult and time-consuming to analyze the source code directly because the concurrency of the components can lead to a state explosion of the system's analysis space. An alternative approach is to create a model of the software, which is smaller and simpler, and analyze the model in lieu of the code.

In prior work, Muscedere's thesis developed a *fact extractor*, **Rex** [15], that builds a model of an existing software system based on facts extracted from the source code's abstract syntax tree (AST). The extracted program model is then imported into a graph database, **Neo4J** [17], to run different analyses expressed in the **Cypher** query language [18]. To perform complex analyses, the Cypher queries can be combined with Awesome Procedures on Cypher (**APOC**) [16] library functions. The general toolchain of the analyses is depicted in Figure 1.1.

A Rex-extracted software model consists of facts about the source code. Consider the C program shown in Figure 1.2. Rex generates a graphical model of this source code where nodes represent program entities (such as functions and variables) and edges represent relationships between program entities (such as calls between functions and assignments between variables). A graphical model of this C program is shown in Figure 1.3. A *varWrite* edge models an assignment between two variables (where one variable is on the left-hand side of the assignment and the other is on the right-hand side). A *call* edge represents a function call. For example, on line 6 of Figure 1.2, variable i is used in the assignment of variable k; therefore in the graphical model, there is a directed edge of type

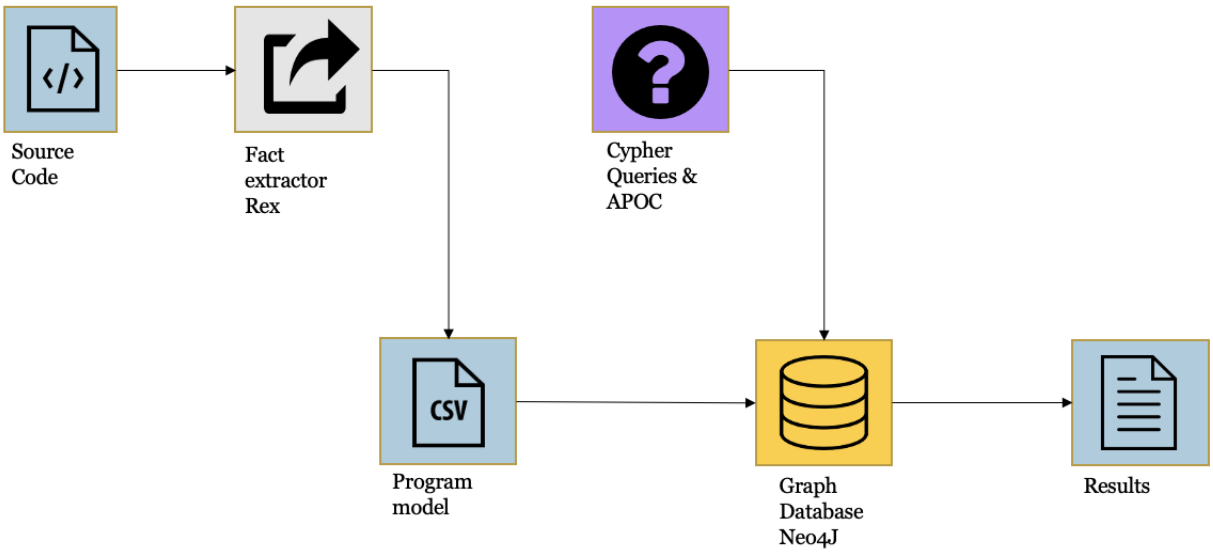


Figure 1.1: Process of analyzing large software systems.

```
1  int goo(int x) {
2      return x * x;
3  }
4
5  int foo(int i, int j) {
6      int k = i * 3;
7      if (k > 5) {
8          int l = goo(k);
9          k = j;
10     } else {
11         int m = k;
12     }
13 }
14
```

Figure 1.2: Small C program

varWrite from node i to node k.

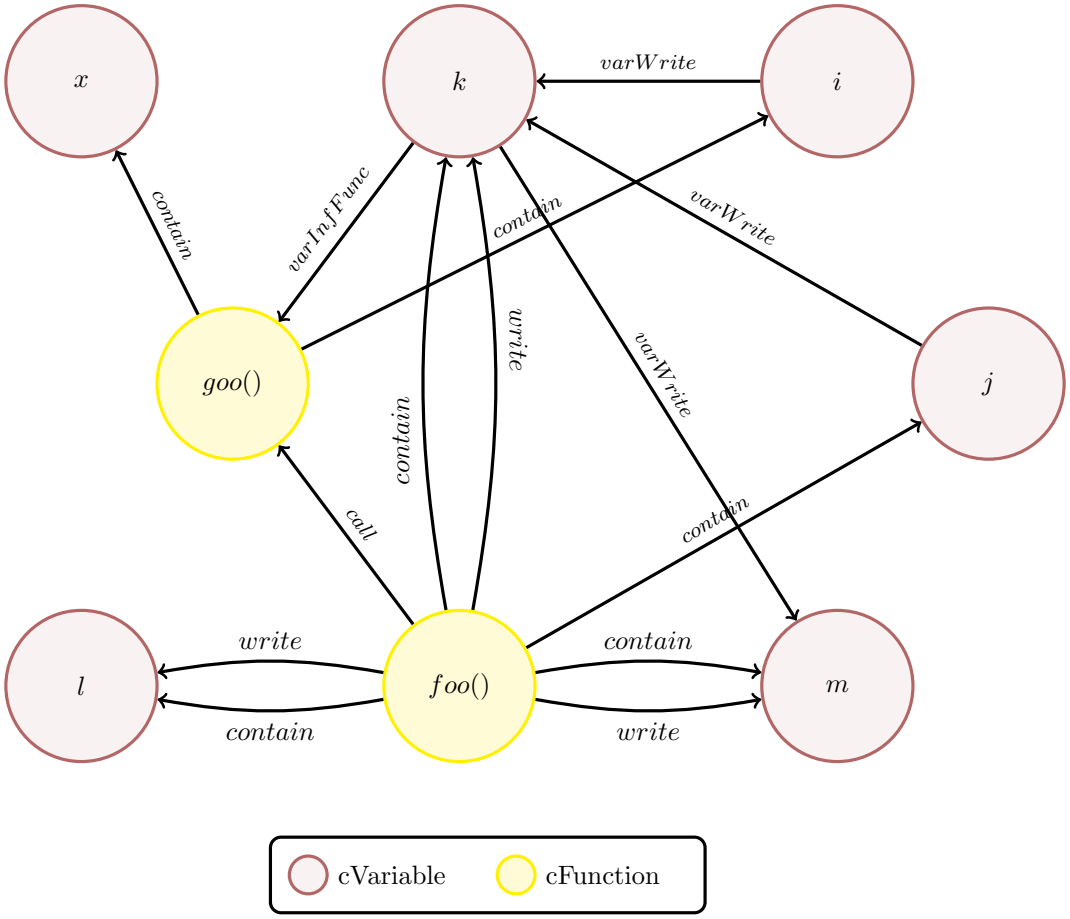


Figure 1.3: Graphical model of small C code example

After importing this graphical model into a graph database such as Neo4J, analyses can be performed using Cypher queries. The Cypher query language enables ad-hoc expression of various analyses such as dataflow and control-flow analyses. For example, a variable-assignment analysis is a simplified dataflow analysis that captures direct and indirect variable assignments in a program. Posing this query of the graphical model in Figure 1.2 produces five dataflow paths, which are listed in Figure 1.4. Lines 1-3 show direct variable assignments in the program. For example, the result on line 1 indicates that variable i affects the value of variable k. Lines 4-5 show indirect variable assignments which means that the first variable in the path indirectly affects the value of the last variable in the path through a series of variable assignments. For example, the result in line 4 indicates that

variable i indirectly affects the value of variable m, through an assignment to k followed by an assignment to m.

Note that the last path in Figure 1.4, which represents an indirect relationship from variable j to variable m, is infeasible. A close inspection of the source code reveals that the assignment from variable j to variable k occurs in the `if-then` block, whereas the assignment from variable k to variable m occurs in the same control statement's `else` block. In any execution of function `foo()`, only one of these statement blocks can be executed, hence the indirect variable relationship from j to m does not hold. This imprecision is caused by limited information in the software model about execution paths in the model's corresponding program.

```
1   i - [varWrite] -> k
2   j - [varWrite] -> k
3   k - [varWrite] -> m
4   i - [varWrite] -> k - [varWrite] -> m
5   j - [varWrite] -> k - [varWrite] -> m
```

Figure 1.4: Results of dataflow cypher query

Muscedere's analysis of an extracted software model [15] includes six different analyses performed on the software model of Autonomoose [27], a self-driving vehicle project. The precision of these analyses ranges from 8.9%-100.0%. The analyses with low precision often produce results that have no corresponding execution paths.

The goal of this thesis is to improve the precision of Cypher-based analyses on Rex-extracted software models. As shown in Muscedere's work, the existing Rex-extracted software model contains a significant amount of invalid results caused by infeasible execution paths. Hence, we need to revise the existing software model and Cypher-based analyses to consider execution path information in order to improve the precision of the analyses results.

The proposed solution described in this thesis is to enrich Rex-extracted software models with additional information about the execution paths of the modeled program. Specifically, we incorporate control-flow-graph (CFG) information about possible execution paths into an extracted program model. Each CFG block is represented as a node in the model, and consecutive CFG blocks are represented as edges connecting a CFG node with its next CFG nodes. Additional graph edges link program entities and relationships with the CFG block(s) in which they occur. This extra information allows query analyses to perform feasibility checks to confirm that analysis results correspond to actual execution paths and

to filter invalid results, thereby improving the precision of analysis results. CFG-based checks can introduce a significant performance overhead because they require analyses of the model's CFG facts and CFG paths. Our approach to improving the performance of analyses with CFG-based checking is to stage the queries using a divide-and-conquer approach that constructs and checks analysis results incrementally. Our work is evaluated on both Robotic Operating System (ROS) and non-ROS software systems.

1.1 Thesis Contribution

The contributions made in this thesis are as follows:

- An extension of Rex with CFG information to model program execution paths. This extension enhances the software model by relating previously extracted facts with the CFG blocks in which they occur, and by recording the relationships between consecutive CFG blocks.
- Revisions of the Neo4J queries and APOC functions to enable on-the-fly CFG-based checking to exclude analysis results that do not correspond to feasible execution paths.
- An evaluation of the precision and performance overhead of a behaviour alteration analysis query with CFG-based checking performed on a ROS-based self-driving vehicle software system, Autonomoose [27]. The CFG-based checking reduces the reported results by 99.77% with an overall overhead of 29.42%.
- An evaluation of the performance overhead of on-the-fly CFG-based analyses performed on multiple open-source ROS-based software systems.
- The application and performance evaluation of a CFG-based behaviour-alteration analysis on a non-ROS-based software controller to evaluate the scalability of our approach.

1.2 Thesis Organization

This thesis is organized as follows. Section 2 provides a brief overview of prior work with respect to the fact extractor Rex and related analyses. Section 3 describes in detail our

extensions to Rex. These extensions include the CFG nodes and edges and associations between existing facts and these CFG nodes. Section 4 provides a description of the changes made to the Neo4J APOC library [16] to support CFG-based checking and changes to query processing to perform CFG-based analyses on software that uses the ROS [19] communication framework. Section 5 reports on the performance and precision evaluations of the CFG-based analyses performed on Autonomoose and a collection of open-source ROS-based projects. Sections 6 and 7 describe the adaptation of our analyses to non-ROS software systems. These two sections also report on a performance evaluation that applies this approach to a 1.5 million line software controller that is non-ROS-based. Section 8 discusses the degree to which our work improves on the precision of our previous model-based analyzer. Section 10 summarizes the work completed in this thesis and suggests areas that can be further investigated.

Chapter 2

Background and Previous Work

In this chapter, we briefly describe concepts that are used remainder of the thesis. We provide backgrounds on fact extractors and the software models they extract, the Neo4J graph database, control-flow-graph-based analyses, and previous analyses performed.

2.1 Robot Operating System

The Robot Operating System (ROS) [19] is an open-source development kit for robot applications. It contains tools and libraries that support hardware abstraction, device drivers, software component communication, visualization, and other common robot design patterns. ROS uses a publisher-subscriber framework shown in Figure 2.1 to enable inter-component communication. Suppose Component A wants to send message m to Component B in a ROS program. First, publisher p is created in Component A and subscriber s is created in Component B. Then, publisher p directly sends message m to topic t which the ROS framework notifies subscriber s about this message. Upon receiving the message, subscriber s calls function f, also known as a **callback function**, to process this message. The main advantage of this publisher-subscriber framework is that it supports many-to-many relationships where the components can send and receive messages from multiple components simultaneously.

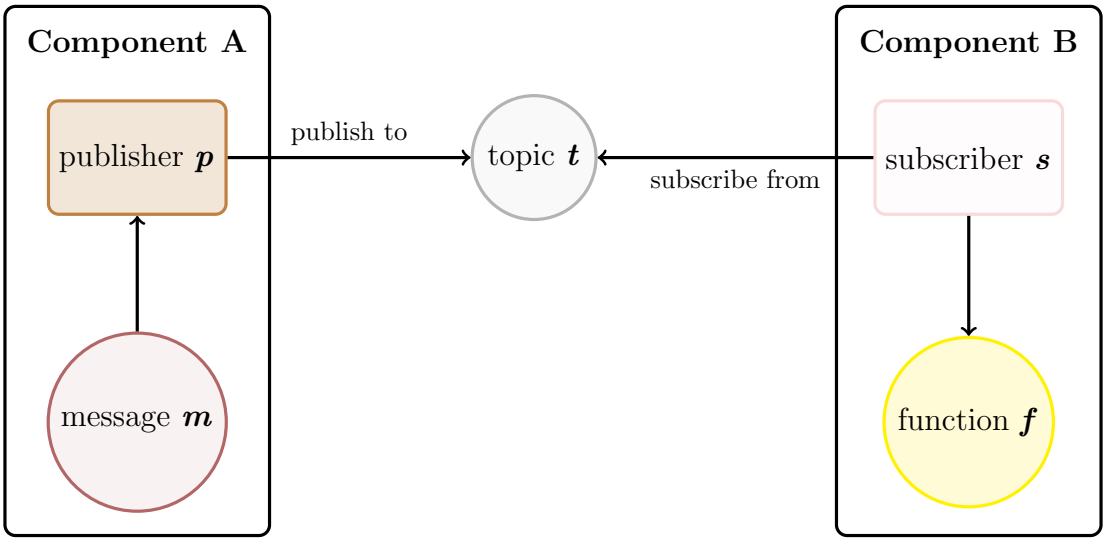


Figure 2.1: Diagram for ROS communication framework

2.2 Rex Fact Extraction and Factbases

Fact extractors are parser-based analyzers that record details from software artifacts. Each extracted detail is known as a **fact**, representing a program entity or describing a relationship between two program entities within the source code. A **factbase** is a collection of facts extracted from a software application.

Rex [15] is a fact extractor specialized for Robot Operating System (ROS) [19] based applications and is implemented on top of the Clang [12] compiler. Rex uses Clang to generate the abstract syntax tree (AST) for a given program fragment, and then Rex walks over the AST to extract facts of interest. The textual representation of the factbase for the C code presented in Figure 1.2 is displayed in Figures 2.2 and 2.3. This factbase contains two large sections: (1) the fact tuples shown in Figure 2.2 represent program entities and relationships between these entities, and (2) the fact attributes shown in Figure 2.3 represent attributes of the corresponding fact tuples. The fact tuple section can be further divided into two subsections: (a) the lines that start with **$INSTANCE** represent the extracted program entities, and (b) the remaining lines list the relationships between these program entities.

In our example C program's factbase, there are eight program entity facts and fourteen relationship facts relating these program entity facts to each other. The different types of program entity facts are as follows:

```
FACT TUPLE :
$INSTANCE "decl;test.c::foo(int,int,)::m;;" cVariable
$INSTANCE "decl;test.c::foo(int,int,)::l;;" cVariable
$INSTANCE "decl;test.c::foo(int,int,)::k;;" cVariable
$INSTANCE "decl;test.c::foo(int,int,)::j;;" cVariable
$INSTANCE "decl;test.c::foo(int,int,)::i;;" cVariable
$INSTANCE "decl;test.c::goo(int,)::x;;" cVariable
$INSTANCE "decl;test.c::foo(int,int,);;" cFunction
$INSTANCE "decl;test.c::goo(int,);;" cFunction
varWrite "decl;test.c::foo(int,int,)::k;;" "decl;test.c::foo(int,int,)::m;;"
varWrite "decl;test.c::foo(int,int,)::i;;" "decl;test.c::foo(int,int,)::k;;"
write "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::k;;"
write "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::l;;"
write "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::m;;"
varWrite "decl;test.c::foo(int,int,)::j;;" "decl;test.c::foo(int,int,)::k;;"
varInfFunc "decl;test.c::foo(int,int,)::k;;" "decl;test.c::goo(int,);;"
contain "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::i;;"
contain "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::j;;"
contain "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::k;;"
contain "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::l;;"
call "decl;test.c::foo(int,int,);;" "decl;test.c::goo(int,);;"
contain "decl;test.c::foo(int,int,);;" "decl;test.c::foo(int,int,)::m;;"
contain "decl;test.c::goo(int,);;" "decl;test.c::goo(int,)::x;;"
```

Figure 2.2: Factbase of Small C program.

```
FACT ATTRIBUTE :
"decl;test.c::foo(int,int,)::i;;" { filename = "/home/ubuntu/paperRex/test.c"}
"decl;test.c::foo(int,int,)::j;;" { filename = "/home/ubuntu/paperRex/test.c"}
"decl;test.c::foo(int,int,)::k;;" { filename = "/home/ubuntu/paperRex/test.c" isControlFlow =
"1" isParam = "0" label = "ubuntu/paperRex/test.c:foo::k" }
"decl;test.c::foo(int,int,)::l;;" { filename = "/home/ubuntu/paperRex/test.c" isControlFlow =
"0" isParam = "0" label = "ubuntu/paperRex/test.c:foo::l" }
"decl;test.c::foo(int,int,)::m;;" { filename = "/home/ubuntu/paperRex/test.c" isControlFlow =
"0" isParam = "0" label = "ubuntu/paperRex/test.c:foo::m" }
"decl;test.c::foo(int,int,);;" { filename = "/home/ubuntu/paperRex/test.c" isCallbackFunc =
"0" label = "ubuntu/paperRex/test.c:foo" }
"decl;test.c::goo(int,)::x;;" { filename = "/home/ubuntu/paperRex/test.c" }
"decl;test.c::goo(int,);;" { filename = "/home/ubuntu/paperRex/test.c" isCallbackFunc = "0"
label = "ubuntu/paperRex/test.c:goo" }
```

Figure 2.3: Factbase of Small C program.

- A **variable** entity fact is created for each function parameter and variable defined in the program (e.g. parameter i and local variable k).
- A **function** entity fact is created for every function definition in the source code (e.g. functions *goo* and *foo*).

In addition to the entity facts, there are six types of relationship facts in the factbase to this example:

- A ***call*** relationship fact represents a function call in the program (e.g. function *foo* calls function *goo*).
- A ***contain*** relationship fact indicates one entity is declared in another entity (e.g. parameter i is defined within function *foo*).
- ***varWrite*** represents a variable assignment (e.g. parameter j is used in the variable assignment of local variable k).
- A ***write*** relationship fact indicates that a function changes the value of a variable (e.g. local variable k is used in an assignment statement in function *foo*).

Table 2.1: Types of Entity Facts

Entity Type	**Description**
cClass	a class definition in a program
cFunction	a function definition in a program
cStruct	a structure definition in a program
cVariable	a variable definition in a program
rosPublisher	a ROS publisher, which sends messages to a specified ROS topic
rosSubscriber	a ROS subscriber, which receives messages that were sent to a specified ROS topic
rosTopic	a ROS topic, which names a message type that can be communicated between components, publishers can "publish" messages to components interested in this topic, and subscribers can "subscribe" to receive all messages of this topic

In general, which facts are of interest depends on the analyses to be performed, but they generally include program entities, relationships between entities, attributes of entities and relationships, and ROS inter-component communication facts. Table 2.1 summarizes the different software entity facts that Rex extracts from C/C++ programs. It includes general entity facts covering variables, functions, classes, and structures. It also includes facts corresponding to ROS communication elements that represent publishers, subscribers, and topics nodes.

Table 2.2: Types of Relationship Facts

Relationship Category	**Relationship Type**	**Description**
Dataflow	(v)-varWrite-(a)	variable **v** is used in the expression of an assignment to variable **a**
Control Flow	(f1)-call-(f2)	function **f1** calls function **f2**
	(v)-varInfFunc-(f)	a variable **v** is used in the condition of a control statement, which affects the calling of another function **f**
	(v)-varInfluence-(p)	a variable **v** is used in the condition of a control statement, which affects whether ROS component **p** publishes a message
Program Structure	(f)-write-(v)	function **f** includes an assignment to variable **v**
	(e1)-contains-(e2)	entity **e1** contains entity **e2** in some form; for instance, a class contains a function
ROS Communication	(p)-publish-(t)	publisher **p** publishes to the ROS topic **t**
	(s)-subscribe-(t)	subscriber **s** subscribes from the ROS topic **t**

Table 2.2 shows the different types of relationship facts that Rex extracts. These facts can be divided into four categories: dataflow facts, control-flow facts, program structure facts, and ROS communication facts. Dataflow facts reflect how data travels through the program through variable assignments (*varWrite*). Control-flow facts represent facts related to control-flow decisions. These facts describe function call relationships (*call*), and variables that influence such function calls through control structures (*varInfFunc* and *varInfluence*). Program structure facts represent location-related facts including which

functions change the values of variables (*write*), and which functions declare which variables (*contain*). The ROS communication facts reflect communication between ROS components, including the notion of subscribing from a topic (*subscribe*) and publishing to a topic (*publish*).

2.3 Neo4J Graph Database, Cypher, and APOC

The facts in an extracted factbase effectively form a graphical model of a program's entities and relationships. Each **node** in the graphical model represents a program entity and the **edges** represent relationships between these entities. Hereafter in this thesis, we use the terms "entity", "relationship", "property" to refer to program elements in the code, and we use the terms "node", "edge", "attribute" to refer to their corresponding facts in the extracted graph model. We can import the factbase into a graph database to visualize and analyze the extracted facts. We use the Neo4J graph database [17].

The analyses are expressed as queries over a Neo4J factbase using the Cypher query language [18]. For example, a simple Cypher query, shown in Figure 2.4, reports all variable-assignment dataflows of length one or more in a given program. Specifically, this query returns the non-reflexive transitive closure of all the variable-assignment facts extracted from the target program. Figure 1.4 from Chapter 1 shows the path returned after running this query on the example C program from Figure 1.2.

To handle more complex queries containing additional filters and decision logic, a Cypher query often makes use of facilities from an open-source plugin called Neo4J Awesome Procedures of Cypher (**APOC**) [16]. APOC is a Java-based plugin that is built on top of the Neo4J 3.5.0 API. It supports the direct manipulation of nodes and relationships in the graphical database and the straightforward implementation of new functions and extensions.

```
1 MATCH vwPaths=(:cVariable)-[:varWrite*1..]->(:cVariable)
2 RETURN vwPaths;
```

Figure 2.4: Dataflow cypher query.

There are two benefits of using a graph database like Neo4J for analyses: (1) analyses are programmable using the query language, rather than being hardcoded and fixed, and (2) Neo4J is optimized for path-based queries, which is ideal for control-flow and dataflow analyses. Our analyses inherit the limitations of Neo4J query language.

2.4 Query Analyses

Muscedere's Master thesis [14] conducted six different feature interaction analyses on the Rex-extracted factbase model of Autonomoose [6], a ROS-based autonomous driving vehicle software system. **Feature interaction analyses** detect potential unexpected behaviours in an automotive software system, resulting from different automotive software components communicating and interacting with each other.

Inter-component-based communication analysis detects direct and indirect communications between two or more components through intracomponent dataflows and ROS inter-component communication.

Loop detection analysis reports instances of direct or indirect self-communications. This type of analysis is the same as inter-component-based communication analysis, except that it checks that the first and last components are the same.

Multiple publishers analysis detects instances when multiple publishers pass messages that lead to the same callback function in a ROS subscriber. Such instances identify publishers that might flood the event handler of a subscriber.

Race condition analysis detects when different publishers attempt to change the same variable in a subscriber component. This may cause nondeterministic behaviour if different orders of input messages lead to different system behaviours. Another potential problem is that the processing of later messages can overwrite the processing of previously received messages.

Behaviour alteration analysis detects potential unexpected behaviours due to the receipt of a message. Specifically, it checks whether the receipt of a ROS message can directly or indirectly influence the calling of a function in a control statement. This type of particular type of function call is seen as a potential behavioural change, initiated in another component that sent the ROS message.

Publisher alteration analysis detects instances where a received ROS message directly or indirectly causes a component to publish a message to a topic. The only difference between this analysis and the behaviour alteration analysis is whether the receipt of the ROS message leads to a potentially unexpected ROS publication or a potentially unexpected function call.

Muscedere's thesis manually evaluated the analysis results and categorized them into: (1) impossible, (2) unlikely, and (3) possible. "Impossible" results are due to errors in the factbase model, "unlikely" results are deemed to be likely false positives, and "probable" results are deemed to likely identify unexpected interactions. To calculate the precision, the

impossible and unlikely instances are treated as false positives and probable instances as true positives. Table 2.3 reports the precision of the six analyses presented in Muscedere's thesis [14].

Table 2.3: Precision of Feature Interaction Analyses

Analysis Type	**False Positive**		**True Positive**	**Precision**
	Impossible	**Unlikely**	**Probable**	
Inter-Component-Based Communication	0	0	15	100.0%
Loop Detection	0	0	2	100.0%
Multiple Publishers	0	0	0	-
Race Conditions	8	1	2	18.2%
Behaviour Alteration	1,021	225	122	8.9%
Publisher Alteration	12	27	25	39.1%

2.5 Control Flow Graph

A control flow graph is a directed graph that models the execution paths in a program [9]. Each node in this control flow graph represents a basic code block (in a program), containing the maximum sequence of consecutive code statements without branching and halting before reaching the end of the block. Each directed edge in the graph represents a successor relationship between two nodes. Branching occurs when a control flow block has two or more successor blocks, and this typically occurs after the decision condition of a control statement.

Control flow graphs are often used for dataflow analyses and reachability and availability analyses for variables and expressions [9]. A data dependency graph [9, 22] is an application of a control flow graph, where each edge in the graph represents a valid pair of variable definition and read access of that variable in the program; the definitions must occur before the read access. Some code-based applications of control flow graphs include compiler optimization [2], iterative dataflow analysis [9, 22], dynamic program slicing [1], alias analysis [11], and the development of other program-related graphs.

Clang [12] recursively walks through the AST to generate the control flow graph (CFG) blocks of a program. Figure 2.5 shows the CFG blocks generated for the C program from Figure 1.2. As observed, each function has an entry block, an exit block, and a set of intermediate blocks. Clang generates three CFG blocks for the functions `goo()` and

```
int goo(int x)
 [B2 (ENTRY)]
   Succs (1): B1

 [B1]
   1: x
   2: [B1.1] (ImplicitCastExpr, LValueToRValue, int)
   3: x
   4: [B1.3] (ImplicitCastExpr, LValueToRValue, int)
   5: [B1.2] * [B1.4]
   6: return [B1.5];
   Preds (1): B2
   Succs (1): B0

 [B0 (EXIT)]
   Preds (1): B1

void foo(int i, int j)
 [B4 (ENTRY)]
   Succs (1): B3

 [B1]
   1: k
   2: [B1.1] (ImplicitCastExpr, LValueToRValue, int)
   3: int m = k;
   Preds (1): B3
   Succs (1): B0

 [B2]
   1: goo
   2: [B2.1] (ImplicitCastExpr, FunctionToPointerDecay, int (*)(int))
   3: k
   4: [B2.3] (ImplicitCastExpr, LValueToRValue, int)
   5: [B2.2]([B2.4])
   6: int l = goo(k);
   7: j
   8: [B2.7] (ImplicitCastExpr, LValueToRValue, int)
   9: k
  10: [B2.9] = [B2.8]
   Preds (1): B3
   Succs (1): B0

 [B3]
   1: i
   2: [B3.1] (ImplicitCastExpr, LValueToRValue, int)
   3: 3
   4: [B3.2] * [B3.3]
   5: int k = i * 3;
   6: k
   7: [B3.6] (ImplicitCastExpr, LValueToRValue, int)
   8: 5
   9: [B3.7] > [B3.8]
   T: if [B3.9]
   Preds (1): B4
   Succs (2): B2 B1

 [B0 (EXIT)]
   Preds (2): B1 B2
```

Figure 2.5: Basic blocks of C program.

five CFG blocks for the function `foo()`. Each intermediate CFG block contains a set of consecutive statements and identifiers of possible predecessors and successors blocks. Entry and exit blocks are similar to immediate basic blocks with the exception that entry blocks have only successor information and exit blocks have only predecessor information. The information in these CFG blocks can be used to create execution paths in the program, and they can be used as filtering conditions for previous analyses to exclude results associated with infeasible execution paths.

Chapter 3

Extensions to Rex

Muscedere's prior work on extending a factbase model from a program's source code reported numerous false positive results [15]. In this chapter, we describe several extensions to our fact extraction to improve the precision of our factbase analyses. This improvement is achieved by testing that reported analysis results pertain to actual execution paths in the code.

3.1 parWrite

The previous fact extractor did not capture dataflows via parameter passing. Instead, it conservatively assumed that a function call was a proxy for dataflows of all possible variables from the calling context to the called function. This assumption was a major source of imprecision in dataflows. This problem is resolved by extracting a new type of relationship fact, a *parWrite* fact, that models the specific dataflow via parameter passing from an actual parameter to a formal parameter. This new *parWrite* fact refines the *call* and *write* facts, in Muscedere's analyses, which reflect a function call followed by the assignment of variables in the called context.

3.2 cReturn

The previous fact extraction also omitted dataflows via function returns. Without considering the dataflow from the called context back to the calling context, some valid dataflow

results may not be reported. We resolved this problem by creating a *cReturn* fact for each function that has at least one `return` statement; this fact represents the value returned from a call to a non-void function. For each variable that appears in a `return` statement expression, a new *varWrite* fact is created to reflect that that variable affects the *cReturn* node's value (a dataflow).

In addition, several relationship facts are generated for every function call that occurs in an assignment statement (*varWrite*), return statement (*retWrite*), nested function call (*parWrite*), and the condition of a control statement where the body has a function call(*varInfFunc*) or a publish statement(*varInfluence*). The types of these new facts related with *cReturn* node are described in Table 3.1.

Table 3.1: Types of *cReturn* Relationship Facts

Relationship Category	**Relationship Type**	**Description**
Dataflow	(v)-varWrite-(r)	variable **v** is used in the expression of a function return statement **r**
	(r)-parWrite-(p)	return node **r** is is passed to parameter **p** through a nested function call
	(r1)-retWrite-(a) (r1)-retWrite-(r2)	function return value **r1** appears either in the expression of an assignment to variable **a** or in another function return statement **r2**
Control Flow	(r)-varInfFunc-(f)	a function with a return node **r** is used in the condition of a control statement, which affects the calling of another function **f**
	(r)-varInfluence-(p)	a function with a return node **r** is used in the condition of a control statement, which affects whether ROS component **p** publishes a message

3.3 pubVar and pubTarget

pubVar and *pubTarget* facts serve a similar purpose as *parWrite* facts. In previous work, it was assumed that any variable assignments in the callback function of a ROS subscriber are influenced by the ROS message received. However, this may not be true because the variable could be assigned to other values that are irrelevant to the ROS message. Furthermore, the dataflow from a ROS message variable to its respective ROS topic was not captured in the factbase, hence was impossible to construct a complete dataflow path from a ROS publisher to a ROS subscriber. We resolved these imprecisions by introducing two additional relationship facts: *pubVar*, which indicates a specific variable is published to the designated ROS topic; and *pubTarget* which maps a ROS topic to a specific parameter of each **callback** function that reacts to the ROS message.

The purpose of a **callback** function in a ROS subscriber is to extract the data from a ROS message and store it in either an attribute or local variable before processing. Thus, a full ROS communication entails: (1) the publisher component creates a ROS message m with a variable assignment from v_0, (2) this ROS message is sent to ROS topic t, (3) the subscriber(s) of the ROS topic receives the message and stores a reference of it to the callback function's parameter p, and (4) the data inside the ROS message (in parameter p) is parsed and stored in one or more local variables or attributes v_1 for additional operations.

3.4 Add File Name to Node Identifier

In previous work, the unique identifier of each node entity in a factbase was the program entity name prefaced by the program entity's parent object or function name. However, this naming convention causes imprecision when multiple objects or functions have the same name but are declared in different locations because the facts related to same-named objects and functions would be merged when instead they should be treated differently. For example, suppose there are two *main* functions defined in two different C files. The original Rex extractor would create a single main function entity node instead of two nodes and all relationship facts related to these two functions would be combined. Analyses of such a software model may return false positive results because the two functions should have distinct contexts. To resolve this imprecision, we revised Rex to embed file names into the identifier to support the differentiation of these objects and functions. This change improves the precision of the factbase and it is easier for engineers to review analysis results.

3.5 Variables and Default Topic Names

Another limitation discussed in previous work is that the topic names of subscribers and publishers must be strings and cannot be variables. This assumption causes the ROS communication elements and relationships involving variable topic names to be excluded from the factbase. Upon further inspection, it is revealed that most topic variables often have default string values, and they are modified based on different configuration options. Therefore, it is still feasible to perform analyses on default communication paths using static analysis techniques. Hence, Rex is extended to map these variables and their default string values, enabling the creation of the corresponding ROS subscriber and publisher nodes.

3.6 call Facts

In addition to the limited dataflow issue in previous work, described in earlier subsections of this chapter, facts about control flows between functions and ROS publishers/subscribers were missing or difficult to infer. Specifically, previous work represents a callback function only as an attribute of a ROS subscriber node. To resolve this limitation, we extend the *call* facts to include two additional variants. The first *call* variant relates a function f to ROS publisher p, representing that function f belongs to ROS component p and that f contains a `publish` statement from p. The second *call* variant relates a ROS subscriber s to its callback function f. In the latter case, the direct relationship between subscribers and their callback functions enables faster queries over multiple components than if callback functions are modeled as attributes of subscriber nodes.

3.7 CFG Facts

The greatest source of imprecision in our previous work on the factbase analyses are analysis results that do not correspond to feasible execution paths. We presented a simple example in Chapter 1 that reported an invalid dataflow between two unconnected basic code blocks in a small program. We propose to augment the Rex factbase with information about control flow graph (CFG) blocks (basic blocks of program statements) and the program facts that hold in each CFG block. This information is used to check whether dataflow and control-flow analysis results refer to feasible execution paths, as represented by a corresponding path of connected CFG blocks.

We use the Clang abstract syntax tree (AST) library to extract CFG block information from a program's source code [12]. Each CFG block generated by the Clang AST library has a unique block number, contains the code associated with the block, and has links to its possible predecessor and successor CFG blocks. We modified Rex to extract new types of entity and relationship facts to model the CFG blocks of a program:

1. a *cCFGBlock* is an entity fact that represents a CFG block within the program's CFG graph.
2. a *nextCFGBlock* relationship fact relates a CFG block fact with a possible successor CFG block fact.

Additional CFG relationship facts relate existing program facts with their corresponding *cCFGBlock* fact. Some non-CFG relationship facts belong to a single *cCFGBlock* fact (e.g. variable assignments), but some relationship facts are related to two CFG *cCFGBlock* facts (e.g. a function call is related to the CFG blocks of both the called and calling contexts). To schematize the queries involving CFG facts, we devise a naming convention for relationships between existing facts and their corresponding CFG facts:

- Every existing relationship fact (except for containment facts) introduces two new relationship facts: one from its source entity to its source *cCFGBlock* fact and another from its destination entity to its destination *cCFGBlock* fact, even if the source and destination entities' *cCFGBlock* facts are the same.
- The names of these new relationships combine the name of the existing relationship type with "Source" or "Destination", depending on whether the existing relationship's source or destination node is involved.

For example, from a given *call* relationship fact, a *callSource* relationship connects the *call*'s source entity to the *cCFGBlock* fact of the calling context, and a *callDestination* relationship fact connects the *call*'s destination entity to the *cCFGBlock* fact of the start of the called function. As another example, for a given variable assignment *varWrite*, relationship facts *varWriteSource* and *varWriteDestination* connect both the *varWrite*'s source and destination entities to the same *cCFGBlock* fact, where the variable assignment occurs. All of the existing relationship facts and their corresponding CFG facts are summarized in Tables 3.2 to 3.4.

Table 3.2: Summary of CFG Relationship Facts (1)

Relationship Type	**CFG Relationship Type**	**Description**
(f)-call-(g)	(f)-callSource-(cfg)	relates the calling function **f** to the CFG block within **f** where function **g** is called
	(g)-callDestination-(cfg)	relates the called function **g** to the entry CFG block of function **g**
(v1)-varWrite-(v2)	(v1)-varWriteSource-(cfg)	relates the RHS variable **v1** in a variable assignment *varWrite* to the assignment's CFG block
	(v2)-varWriteDestination-(cfg)	relates the LHS variable **v2** of a variable assignment *varWrite* to the assignment's CFG block
(v)-parWrite-(p)	(v)-parWriteSource-(cfg)	relates an actual parameter **v** of a parameter assignment *parWrite* to the CFG block where the function is called
	(p)-parWriteDestination-(cfg)	relates a formal parameter **p** to the entry CFG block of p's function
(r)-retWrite-(v)	(r)-retWriteSource-(cfg)	relates the return statement **r** from a function to the exit CFG block containing **r**
	(v)-retWriteDestination-(cfg)	relates the LHS variable **v** of an assignment, whose RHS includes a function call, to the CFG block of the assignment
(f)-write-(v)	(f)-writeSource-(cfg)	relates a function **f**, that contains an assignment to variable **v**, to the CFG block of the assignment
	(v)-writeDestination-(cfg)	relates a variable **v**, that is being assigned in function **f**, to the CFG block of the assignment

Table 3.3: Summary of CFG Relationship Facts (2)

Relationship Type	**CFG Relationship Type**	**Description**
(p)-publish-(t)	(p)-publishSource-(cfg)	relates a ROS publisher **p** to the CFG block containing the *publish* statement
	(t)-publishDestination-(cfg)	relates a ROS topic **t** to the CFG block containing the *publish* statement
(v)-pubVar-(t)	(v)-pubVarSource-(cfg)	relates a variable **v**, that is being published to a ROS topic, to the CFG block of the *publish* statement
	(t)-pubVarDestination-(cfg)	relates a ROS topic **t**, that is being published, to the CFG block of the *publish* statement
(t)-pubTarget-(p)	(t)-pubTargetSource-(cfg)	relates a topic **t** of a received ROS message to the entry CFG block of the callback function associated with the subscriber
	(p)-pubTargetDestination-(cfg)	relates a parameter **p** of a callback function to the entry CFG block of the callback function associated with the subscriber
(t)-subscribe-(s)	(t)-subscribeSource-(cfg)	relates a ROS topic **t** to the entry CFG block of the callback function associated with the subscriber
	(s)-subscribeDestination-(cfg)	relates a ROS subscriber **s** to the entry CFG block of the callback function associated with the subscriber

Table 3.4: Summary of CFG Relationship Facts (3)

Relationship Type	**CFG Relationship Type**	**Description**
(v)-varInfluence-(p)	(v)-varInfluenceSource-(cfg)	relates a variable **v**, that appears in the condition of a control statement that calls a publisher, to the CFG block of the control statement's decision condition
	(p)-varInfluenceDestination-(cfg)	relates a publisher **p**, that is called in the body of a control statement, to the CFG block of the *publish* statement within the body of the control statement
(v)-varInfFunc-(f)	(v)-varInfFuncSource-(cfg)	relates a variable **v**, that appears in the condition of a control statement that calls a function, to the CFG block of the control statement's decision condition
	(f)-varInfFuncDestination-(cfg)	relates a function **f**, that is called in the body of a control statement, to the CFG block where the function is called within the body of the control statement

Chapter 4

Neo4J Queries with CFG-Based Checking on ROS Software

In the previous chapter, we discuss how we extended the extracted factbase of a program with control-flow graph (CFG) facts that abstractly model programs' execution paths. In this chapter, we discuss how we use these new CFG facts to incrementally construct analysis results, such that each increment is tested and filtered out if there is no corresponding CFG execution path. In addition, we discuss strategies to improve the scalability of the analyses of large software systems.

There are several ways that graphs are used in this thesis:

- The graphical depiction of a program's extracted factbase, where graph nodes represent program entities and graph edges represent relationships between program entities
- The graphical depiction of a query, which effectively represents patterns or subgraphs that the query engine looks for in the factbase
- Analysis results are depicted graphically, where the nodes and edges in a graph represent program elements and their relationships in the factbase. The results can be presented as paths or graphs if the paths share common program elements.

4.1 CFG-Based Checking

Now that we extended our extracted factbase with new CFG facts, we need to revise the Neo4J query processing algorithm to check that the returned results correspond to CFG execution paths in the program. Execution paths are abstractly represented by paths among consecutive CFG blocks. In general, the processing of a Neo4J query incrementally constructs result paths that match the user's query, where each increment appends an edge to a subpath that is a partial result of the query. We modify the analysis queries so that with each increment, the query checks that there is a corresponding execution path from the last edge in the (partial result) subpath to the newly appended edge. That is, we check that there is a sequence of *cCFGBlock* nodes (related by *nextCFGBlock* edges) from a destination *cCFGBlock* node of the last edge in the subpath to a source *cCFGBlock* node of the edge being appended. We call this **CFG-based checking**. We discuss below how CFG-based checking varies depending on the type of edge being appended to a partial result.

4.1.1 varWrite

A *varWrite* fact represents a dataflow relationship from one variable to another through a variable assignment or in a return statement. In a *varWrite* (*x, y*) fact, the righthand side (RHS) of the assignment is an expression that includes variable x, and the lefthand side (LHS) of the assignment is, or includes, variable y; the RHS and LHS of the assignment are in the same CFG block. The graphical pattern of appending a *varWrite* edge to a CFG-checked path is shown in Figure 4.1. The solid directed edges in the top portion of this graph represent the subpath of an already processed partial analysis result, and the edges along the bottom represent the CFG path consisting of CFG block nodes and CFG edges associated with this subpath. To append a *varWrite* edge to this partial result subpath, we check that there also exists a path consisting of zero or more *nextCFGBlock* edges from the last CFG block node (cfg_i) associated with the subpath's endpoint to the source CFG block node (cfg_n) associated with this *varWrite* edge, with no intervening assignments to the LHS variable, as shown by the dashed portion of this graph. A variation of a *varWrite* (*x, r*) fact occurs when a variable x is used in the return statement of a function, with a return node of r. The CFG check of such a variation is the same as the general case mentioned above.

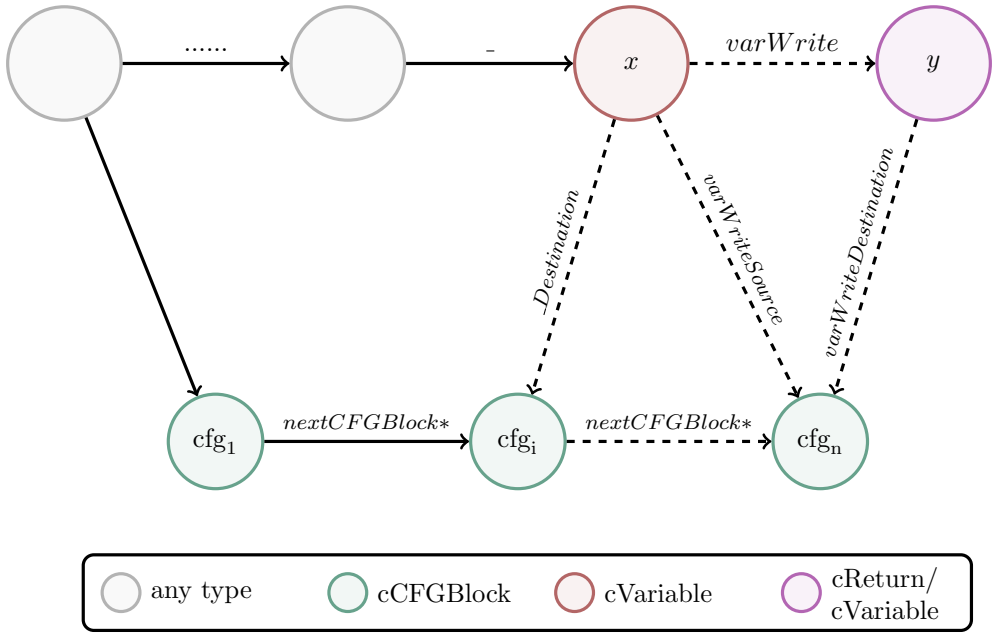


Figure 4.1: Graphical pattern of a subquery involving a *varWrite* fact, and its CFG-based check

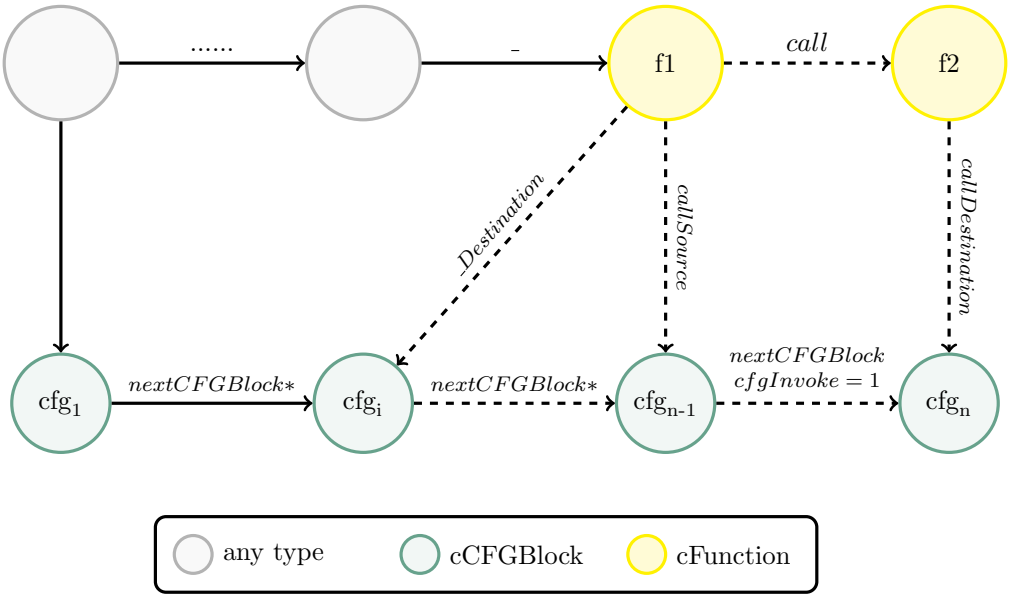


Figure 4.2: Graphical pattern of a subquery involving a *call* fact, and its CFG-based check

4.1.2 call

A *call* fact represents a function call from a calling function to a called function. This fact is different from the *varWrite* fact because the calling function and the called function belong to different CFG blocks. Moreover, their respective CFG blocks are related to each other with a single *nextCFGBlock* fact denoted with the attribute ***cfgInvoke***. Suppose the solid portion of this graph represents an already processed partial query result that is CFG-checked. To append a *call* edge to this CFG-checked subresult, we need to check that there is a sequence of zero or more *nextCFGBlock* edges from the subpath's last CFG block node (cfg_i) to the *call* fact's source CFG node (cfg_{n-1}), as shown in the dashed portion of this figure.

There are two special variations of this *call* fact. The first variation occurs when a function calls a ROS publisher, and the second one occurs when a ROS subscriber calls its callback function. The CFG-based check for the former variations is similar to the CFG check when appending a *varWrite* edge. It first finds the CFG block node containing the publishing statement, then it checks that there exist zero or more *nextCFGBlock* edges between the subpath's last CFG block node to the publish's source CFG block node. See Appendix B for the diagrams of this variation. A *call* relation from the ROS subscriber to its callback function represents an entry point of the component, and no CFG-based

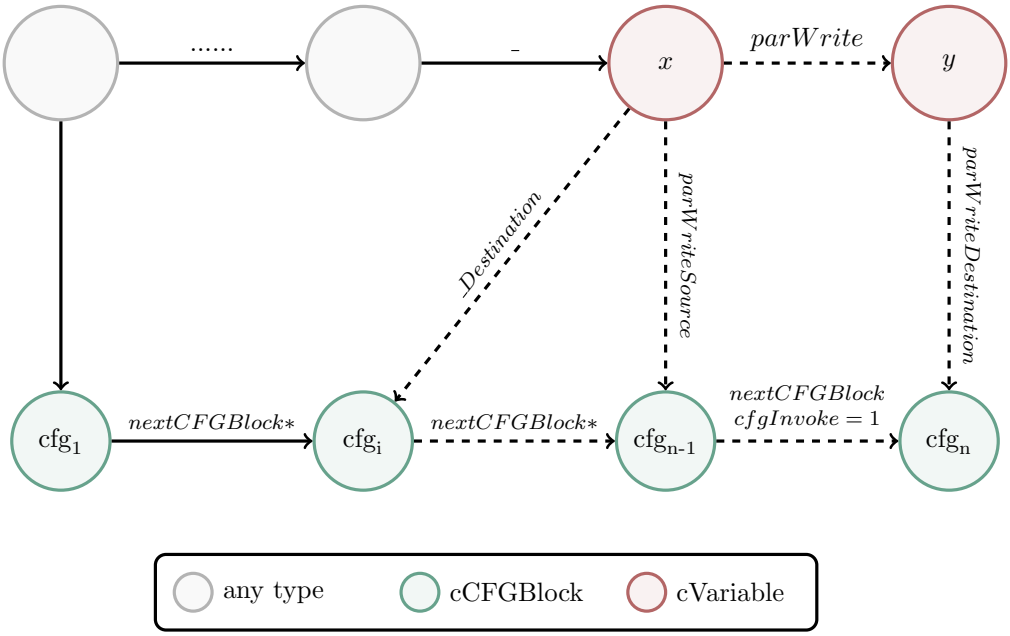


Figure 4.3: Graphical pattern of a subquery involving a *parWrite* fact, and its CFG-based check

checking is involved when appending this variant.

4.1.3 parWrite

A *parWrite* fact represents a parameter being passed from a calling function to a called function. This type of fact is similar to a *call* fact, in that the fact's source and destination CFG blocks are different, but are related with a single *nextCFGBlock* fact, annotated with the attribute ***cfgInvoke***. In addition, a *parWrite* fact reflects a dataflow from actual to formal parameters, whereas a *call* fact involves a function call that may be parameterless. Specifically, *parWrite* (x,y) represents a dataflow from a parameter that is an expression containing variable x to a formal parameter y. To append a *parWrite* edge to a checked subpath, we check that there exists a sequence of zero or more *nextCFGBlock* edges from the subpath's last CFG block node (cfg_i) to the *parWrite*'s source CFG node (cfg_{n-1}), with no intervening assignment to the actual parameter. The dashed portion of Figure 4.3 depicts the relationship to be checked as a part of CFG-based checking.

There is one special variation to a *parWrite* fact: when the actual parameter is a

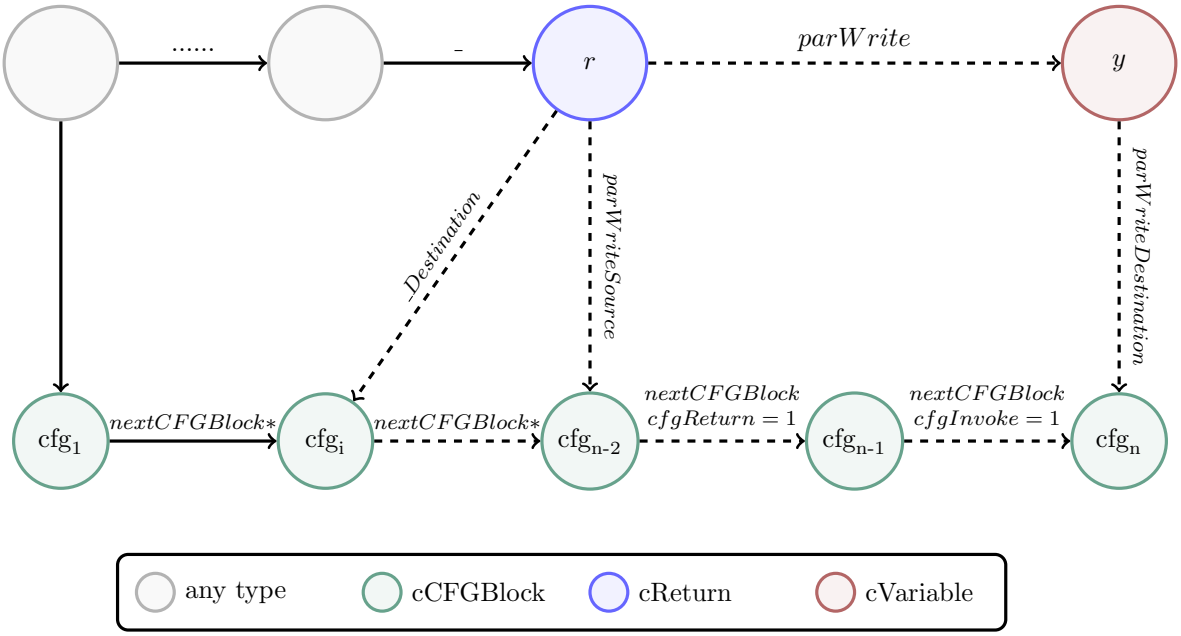


Figure 4.4: Graphical pattern of a subquery involving the special variation of a *parWrite* fact, and its CFG-based check

return node (i.e. the result of a function call). This variation occurs when there is a nested function call, and the value of the inner function is a parameter of the outer function call. The CFG check of such a variation is similar to that of the general case, except that we know there are exactly two *nextCFGBlock* edges between the subpath's CFG last CFG block node and the source CFG block node of the appended fact: the first *nextCFGBlock* edge maps the return statement of the inner function call to that function's exit CFG node, and the second *nextCFGBlock* edge connects that exit CFG node to the inner function's calling context, before passing the returned value to the outer function call. The CFG check of this variation is shown in Figure 4.4.

4.1.4 retWrite

A *retWrite* fact represents a variable assignment that contains a function call with a return value. Specifically, *retWrite* (r,x) records dataflow from a function return value r to a variable x in the calling context. The source CFG node of this fact is the exiting block of the called function and the destination CFG node is the CFG block in the calling context that contains the variable assignment. In addition, the source and destination CFG blocks

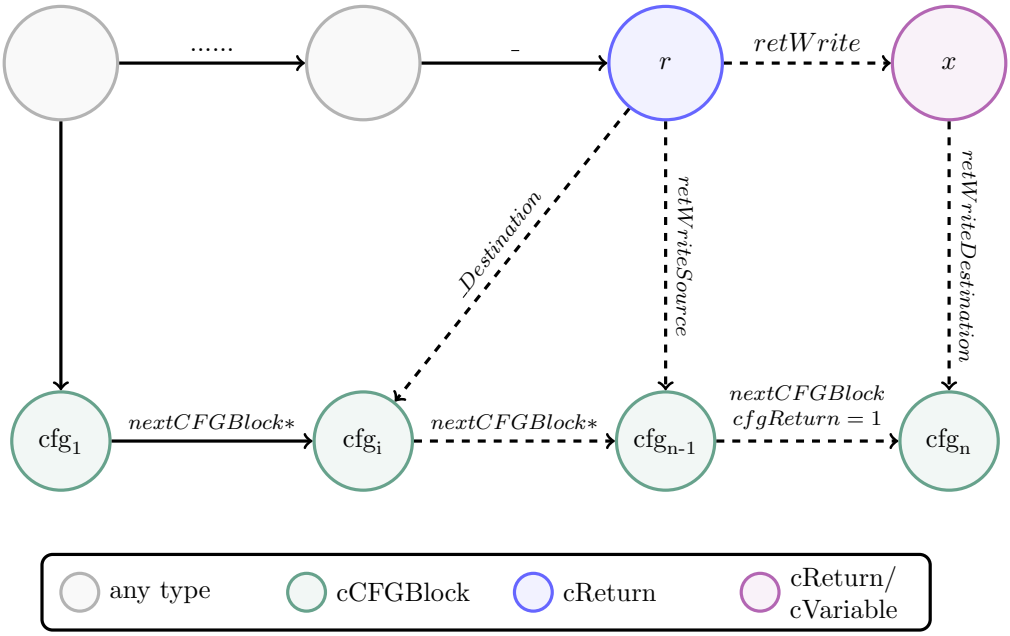


Figure 4.5: Graphical pattern of a subquery involving a *retWrite* fact, and its CFG-based check

are related to each other with a single *nextCFGBlock* fact, annotated with the attribute ***cfgReturn***. To append a *retWrite* edge to the end of a CFG-checked subpath, we check that there exists a sequence of zero or more *nextCFGBlock* edges from the subpath's last CFG block node (cfg_i) to the *retWrite*'s source CFG block node (cfg_{n-1}). The dashed portion of Figure 4.5 depicts the edges to be checked as a part of CFG-based checking. A variation of *retWrite*(*r1, r2*) fact occurs when a function with a return value *r1* is called in the return statement of the callee function with a return value *r2*. The CFG check of appending such variation is the same as the general case.

4.1.5 write

A *write* fact represents a variable assignment contained within a function. The CFG-based checking for appending a *write* edge to the end of a subpath is similar to appending a *varWrite* edge to a subpath because the function and the assigned variable are associated with the same CFG block that the assignment occurred in. See Appendix B for the diagram.

4.1.6 publish

A *publish* fact is created when a ROS publisher publishes to a ROS topic. The source and destination CFG block of a *publish* fact both refer to the one in which the *publish* statement occurs. Thus, to append a *publish* edge to an already-checked subpath, we perform a similar check as for appending a *varWrite* edge. See Appendix B for the diagram.

4.1.7 pubVar

A *pubVar* fact represents a ROS message variable being published to a ROS topic. The fact's source and the destination CFG blocks are the same: they both refer to the CFG block in which the ROS publish function is called. Thus, to append such a fact to a CFG-checked subpath, we perform a CFG-based check that is the same as for a *varWrite* edge, except that we do not perform an intervening check (i.e. that the ROS message variable has not been overwritten before publishing). The reason the intervening check is omitted on the latter is that the ROS message object contains multiple attributes that need to be modified before publishing this message. These ROS message attribute assignments represent internal field changes that cannot be tracked by Rex because it may cause a path explosion when considering these internal field changes.Hence, to avoid filtering valid results, we do not perform this intervening check when appending a *pubVar* edge. See Appendix B for the diagram.

4.1.8 pubTarget

A *pubTarget* fact is created when a ROS message is received and stored as the subscriber's callback function's parameter. Both the source and destination CFG blocks are the entry CFG block of the callback function. Rex performs separate compilation/fact-extraction of components, thus it does not extract CFG edges between ROS components. CFG paths in publisher components end at a `publish` command, and they resume in a subscriber component at a `subscribe` command. Thus, there is no CFG-based checking when appending *pubTarget* edge.

4.1.9 subscribe

A *subscribe* fact represents a ROS component that subscribes to a ROS topic. Both the source and destination CFG blocks of this fact are the same entry CFG block of the callback

function. There is no CFG-based checking when appending a *subscribe* fact because this fact represents a control flow entry to a new component. As mentioned above, Rex conducts separate compilation/fact extraction for each ROS component and it extracts no CFG edges between components. Thus, there is no CFG edge from the publisher component to the subscriber component to check, but the CFG nodes will still be used for determining feasibility when appending to this relationship fact.

4.1.10 varInfFunc

A *varInfFunc* fact represents a variable that appears in the decision condition of a control statement, whose body contains a function call. The source CFG block is different from the destination CFG block because the *varInfFunc* source node (the variable) is in the CFG block containing the decision condition, and the *varInfFunc* destination block (the function call) is in some CFG block within the body of the control statement. The exact number of *nextCFGBlock* relations between the CFG source and destination block is unknown. We can only conclude that it is either one or more. To append a *varInfFunc* edge to an already-checked subpath, we check that there exists a sequence of one or more *nextCFGBlock* edges from the subpath's last CFG block node (cfg_i) to the *varInfFunc*'s source CFG block node (cfg_i), with no intervening assignment to the decision variable. In addition, we need to check that there is a path with one or more *nextCFGBlock* edges from the source CFG node (cfg_i) associated with the *varInfFunc*'s decision condition, and the CFG block node (cfg_n), associated with the function call. The dashed part of Figure 4.6 depicts a graphical pattern of the edges to be checked as a part of CFG-based checking.

As a variation of a *varInfFunc* fact, a function call with a return value could be part of the decision condition. The CFG-based check for this variant is similar to the check for the general case, except that we know there are exactly two *nextCFGBlock* edges between the last CFG block node of the subpath and this decision condition because the last edge in the checked subpath must be a *varWrite* to a return node: (1) the *nextCFGBlock* edge from the return statement's CFG block node to the exit CFG block node of the called function, and (2) the *nextCFGBlock* edge from this exit node to the CFG block node of the *varInfFunc*'s decision condition. See Appendix B for the complete diagram of this variation.

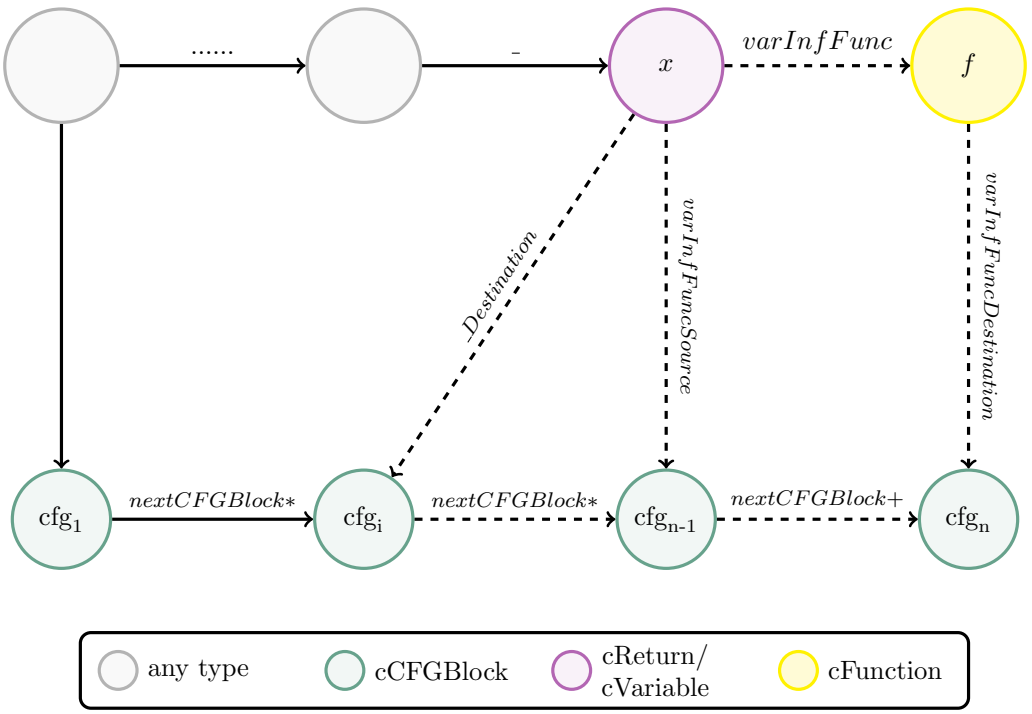


Figure 4.6: Graphical pattern of a subquery involving a *varInfFunc* fact, and its CFG-based check

4.1.11 varInfluence

A *varInfluence* fact is a variation of a *varInfFunc*, in which the variable in the decision condition of a control statement influences whether a ROS publish command (rather than a function call) within the control structure is executed. The source and destination CFG blocks are the same as in *varInfFunc*, where we know only that they are connected by one or more *nextCFGBlock* relations. To append this *varInfluence* edge to an already-checked subpath, we perform the same check as for a *varInfFunc* edge. There is also one variation of this fact, that occurs when a function call with a return value could be part of the decision condition. The CFG check of this variation is the same as the one for *varInfFunc*. See Appendix B for the complete diagrams of both cases.

4.2 Scalability

Performing complex queries on a large factbase can take significant time and resources. By extending the factbase and queries to include CFG-based checking, we further increase resource consumption because each fact being appended to a partial result needs to be checked. Moreover, the end of a partial-result path and the facts to be appended can both be associated with multiple CFG block nodes, thus a CFG-based check effectively needs to find *all* corresponding CFG paths from each CFG block node associated with the end node of the partial-result path to each CFG block node associated with the source node of the appended edge.

To get a sense of the scalability problem, one unoptimized query performed on a non-trivial factbase (extracted from a codebase consisting of 18,738 lines of code) was unable to finish after two weeks. The unsatisfactory performance motivated the development of three techniques to improve the efficiency of query processing: (1) on-the-fly CFG-based checking, (2) reporting shortest-path results only, and (3) two-phased queries.

4.2.1 On-the-Fly Checking

The naïve approach to performing CFG-based checking would be to perform an analysis without the checking and then as a post-processing step filter out the results that fail a CFG-based feasibility test. However, performing CFG-based checking on all results is expensive because a high percentage of returned results are infeasible (i.e., have no corresponding execution path). Thus, the database engine expends considerable resources

exploring and extending invalid results only to have them filtered out in the post-processing phase.

Instead, we devised an on-the-fly checking technique that incorporates the CFG-based checking into the analysis query. Specifically, we revise the query processing engine to check every extension to a partial result: every time an edge is added to the end of a partial-result path, there is a check for a corresponding CFG path from a destination CFG block node associated with the partial path's last relationship to the source CFG block node of the extension. If there is no such path, then the partial result is not extended and exploration of the extended path is terminated.

4.2.2 Shortest Paths

In addition to the on-the-fly approach, we revised the path-based queries to report *all shortest-path results* with the same endpoints instead of *all result paths*. This query design decision is made to reduce resource consumption, with the assumption that engineers are interested in an example of each query result and that further instances of the same result are of decreasing value. We use a modified version of the all-shortest paths algorithm [13] in our reasoning engine because there is one case where we need to return all results: it is impossible to disambiguate the calling context of a function return in a static analysis. Hence, if a query result includes a function return, our "all shortest path" algorithm will include *all* edges that relate the function return to all possible calling contexts, to prevent valid paths from being excluded from the reported results.

4.2.3 Two-Phased Queries (ROS)

Analysis results span across multiple components, and many results contain common subpaths within components. To reduce redundancies in checking for the same subpaths within distinct longer results, we propose a two-phased query analysis that reduces the overall performance by dividing an analysis into a set of simpler queries over smaller versions of the factbases. In the first phase, we partition the system-wide factbase into component factbases and perform component-based queries. After obtaining the component-based sub-results (e.g., representing intracomponent dataflows), a summary relationship fact is created for each intracomponent sub-result and is added to the full factbase. In the second phase, a system-wide query is performed over the full factbase augmented with the new summary facts to obtain the inter-component sub-results. Lastly, we replace the summary links in the system-wide query results with their corresponding intracomponent sub-results

to construct the full query results over the entire factbase. This approach works only when (1) the factbase can be partitioned into multiple smaller factbases associated with different software components; and (2) the queries have well-defined entry and exit points in each component.

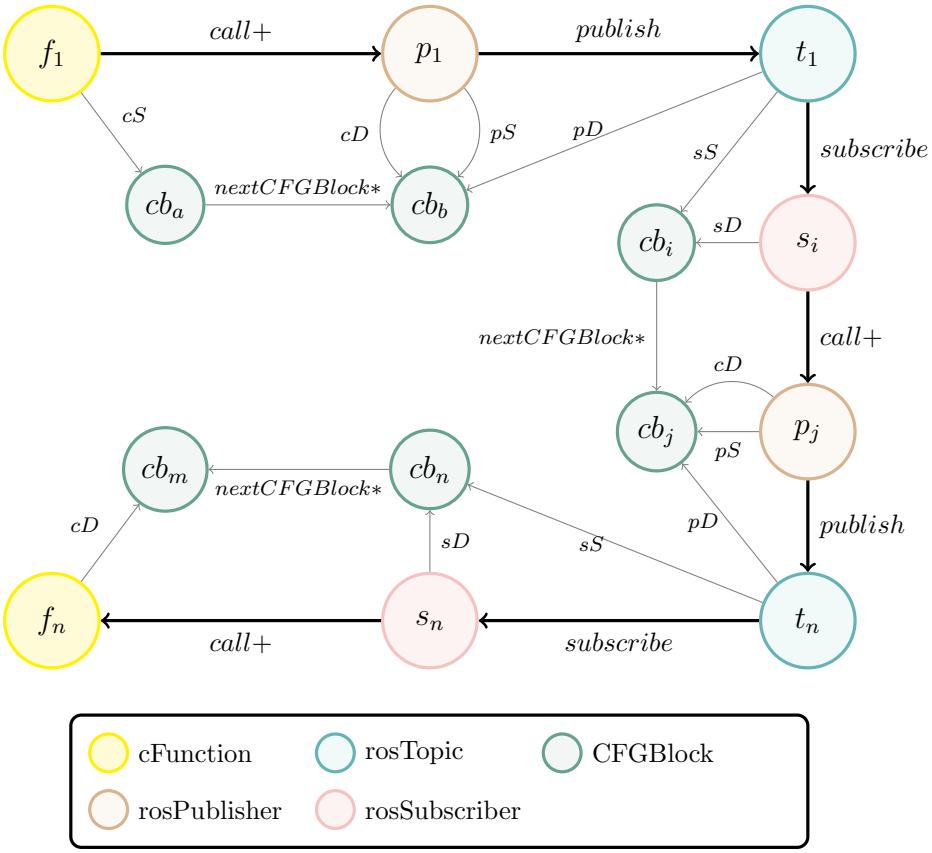
Looking at the process in more detail, consider the graphical pattern of a cross-component call-graph query shown in Figure 4.7. We first create a factbase for each component by running Rex on each of the components' respective code files. Each component factbase will include inter-component facts that ROS publishers send messages to ROS topics (*publish*) and that ROS subscribers receive messages from ROS topics (*subscribe*).

Each query is divided at the point of ROS communications between components. For example, the call-graph query is divided into three intracomponent subqueries:

- a prefix subquery: identifies paths of function calls, ending with a call to a ROS publisher publishing to a ROS topic.
- an intermediate call-graph subquery: identifies paths of function calls in an intermediate component. A call path starts when the component (a ROS subscriber) receives a ROS message associated with a ROS topic, and it ends when the component (also a ROS publisher) publishes to another ROS topic.
- a suffix subquery: matches the termination condition of a call-graph query. A call path that starts when the component (a ROS subscriber) receives a ROS message associated with a ROS topic, and it ends with a call to a function that has no internal function calls.

All three subqueries are performed on every component's factbase to generate intra-component results. We add to the full factbase a new summary relationship fact for each of these intracomponent call-graph results. The outcome is an augmented system-wide factbase that contains the facts extracted from all components plus the newly-added summary facts that represent intracomponent sub-results. We then perform a system-wide query over the augmented factbase to find all cross-component call paths from a function in one component to an indirectly called function in another component. Lastly, we construct the final cross-component call-graph results by substituting the summary facts that appear in a system-wide query result with their corresponding intracomponent call path. Figure 4.8 displays the patterns for the intracomponent subqueries and the overall system-wide query.

We hypothesize that two-phased queries are more efficient because many query results contain the same intercomponent sub-paths. The two-phase queries eliminate repeated



cS : callSource
cD : callDestination
sS : subscribeSource
sD : subscribeDestination
pS : publishSource
pD : publishDestination

Figure 4.7: The graphical query pattern for an unoptimized cross-component call-graph analysis (for ROS applications)

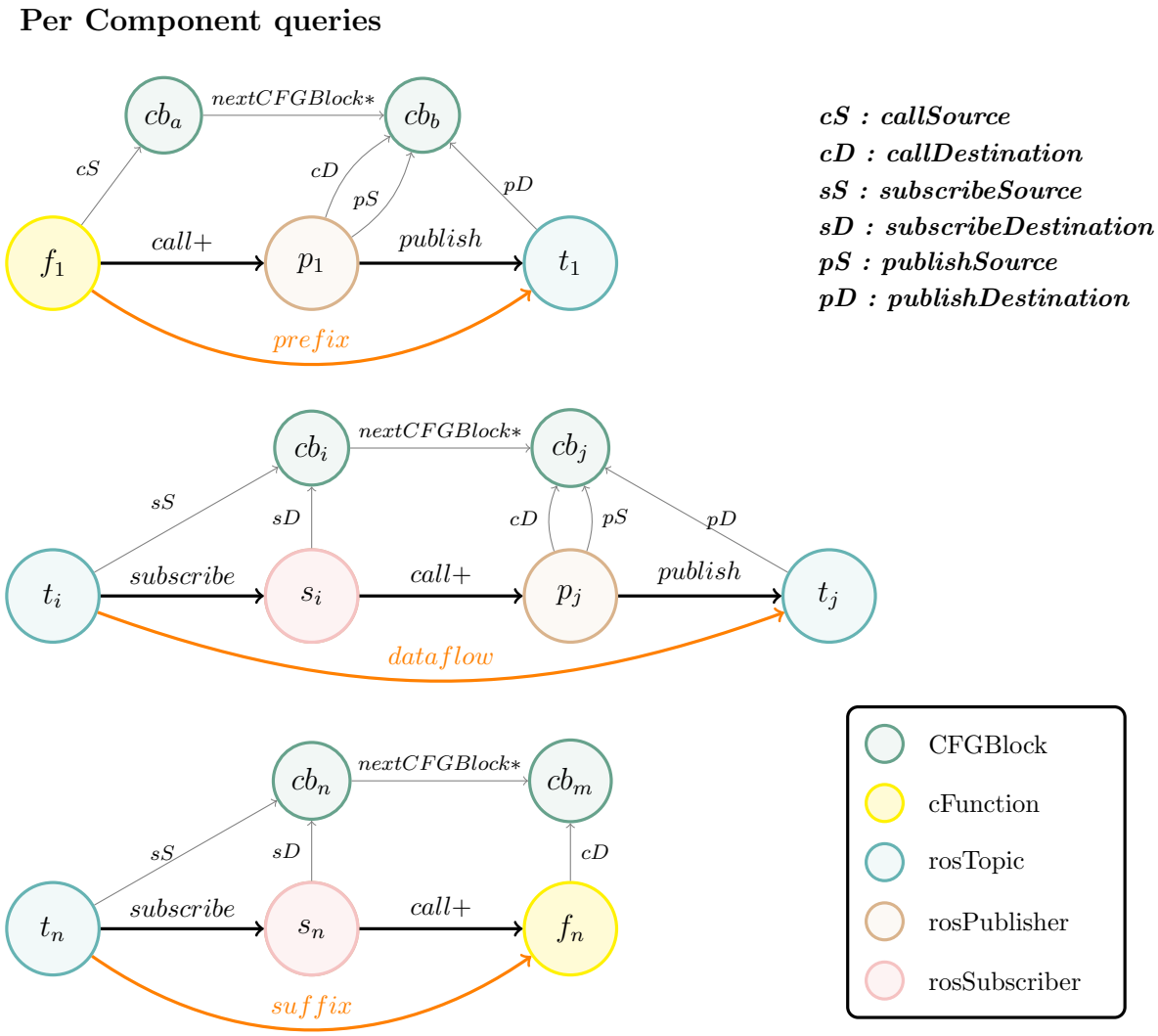


Figure 4.8: The graphical pattern for an incremental cross-component call graph query. The top half of the figure depicts the intracomponent query patterns and the bottom half of the figure depicts the system-wide and inter-component query pattern (for ROS applications)

searches for the same intracomponent paths that appear in multiple full-system results, thereby reducing the time to complete the queries.

4.3 Neo4J Queries and APOC

Neo4J [17] is the graph database we use to perform queries. It allows us to find and visualize query results of interest.

The power of the Cypher query language is limited as it supports relatively simple search tasks, whereas the CFG-based checking we want to perform requires greater control of the underlying searching algorithm and factbase. Thus we employ and extend the **Awesome Procedures on Cypher (APOC)** [16] library to achieve the expressity we need to complete the CFG-based checking task. APOC is an open-source Github project under the Apache license. It contains a variety of custom functions and procedures that perform complex Neo4J Cypher tasks.

We have extended APOC with a new function to perform CFG-based checking on ROS-based systems. This function returns a CFG-checked path meeting all requirements and options specified by the user. It returns all shortest-path results, thereby avoiding the additional overheads incurred by finding all possible paths but uses the modified version of the shortest paths algorithm described in Section 4.2.2. The APOC function that performs CFG-based checking on queries of ROS applications is shown below; all parameters within curly brackets are optional settings. The modified APOC is available in the forked repository [1].

apoc.cfgPath.ROSPath(start, {endE, endN, cfg, relSeq, repeat, shortest, config, filter})

- *start*: a starting graph element (can be a node or an edge); cannot be null
- *endE*: an ending edge. The default value is null
- *endN*: an ending node. The default value is null
- *cfg*: a boolean value indicating whether or not to perform CFG-based checking. Default value is *true*
- *relSeq*: the pattern of edges that defines the analysis query

[1] APOC forked Github Repository: https://github.com/xyfke/neo4j-apoc-CFG-SE

- *repeat*: a boolean value indicating whether or not the *relSeq* identified above is allowed to be repeated. Default value is *false*
- *shortest*: a boolean value indicating whether or not the analysis uses our shortest path function (described in Section 4.2.2)
- *config*: a dictionary that describes each edge type in *relSeq*, the pattern of *nextCFGBlock* edges connecting the relationship's source and destination CFG nodes
- *filter*: the types of CFG nodes allowed in the CFG-checked analysis results

The component-based and system-wide Cypher/APOC queries for a cross-component call-graph analysis are shown in Figure 4.9 to 4.12. Figure 4.9 shows the graphical pattern of a prefix subquery for an intracomponent call-graph analysis. The `MATCH` statements in Lines 1 and 2 retrieve all possible starting facts (any function nodes) and ending facts (i.e. any publishing edges) in the component factbase. For each pair of start and end facts, it uses the *apoc.cfgPath.rosFindPaths* function to return CFG-checked call paths within the factbase. In Figure 4.9, *f1* is passed as a *start* node and *pub* is passed as an *endE*. The *relSeq* parameter in Line 4 specifies the accepted relations in the result paths, which is one or more *call* relationships; and the *filter* parameter in Line 5 serves as a node filter, indicating that only *function* and ROS *publisher* nodes are accepted when constructing the query results. The *config* describes the pattern between every *call* relation's CFG nodes: Lines 7-10 indicate that every *call* relationship from a function node to another function node involves two CFG blocks, which are joined together by a single *nextCFGBlock* relation with an attribute of *cfgInvoke*. After running the APOC query, the results are returned as a list of paths, and Lines 12-13 retrieve each path from the list and print it out in the console of Neo4J.

Figures 4.10 and 4.11 present the graphical patterns of an intermediate call-graph subquery and a suffix call-graph subquery, respectively. The parameters used in the APOC functions are essentially the same, except that both queries start with a *subscribe* edge, and the suffix call-graph subquery result terminates with a function *call* whose destination function has no internal function call.

Lastly, Figure 4.12 shows the graphical pattern of a system-wide call-graph query performed on the augmented system-wide factbase which includes summary facts produced by the intracomponent call-graph subqueries. The resulting paths are sorted based on the number of components involved in each path, with results involving the highest number of components reported first.

```
// Retrieve all pairs of starting subscribe edge and ending publish edge of the
↪  intermediate inter-component call graph query
MATCH (f1:cFunction)
MATCH (:rosPublisher)-[pub:publish]->(:rosTopic)
// Find all shortest paths consisting of one or more call edges for each pair of
↪  start and end edges
WITH *, apoc.cfgPath.rosFindPaths(f1, {
    relSeq : "call+",
    filter : "cFunction,rosPublisher",
    endE : pub,
    config : [
        {name : "call", startLabel : "cFunction", endLabel : "cFunction",
        attribute : "cfgInvoke", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure 4.9: Prefix query for ROS component call-graph.

```
// Retrieve all pairs of starting subscribe edge and ending publish edge of the
↪  intermediate inter-component call graph query
MATCH (:rosTopic)-[sub:subscribe]->(:rosSubscriber)
MATCH (:rosPublisher)-[pub:publish]->(:rosTopic)
// Find all shortest paths consisting of one or more call edges for each pair of
↪  start and end edges
WITH *, apoc.cfgPath.rosFindPaths(sub, {
    relSeq : "call+",
    filter : "cFunction,rosPublisher",
    endE : pub,
    config : [
        {name : "call", startLabel : "cFunction", endLabel : "cFunction",
        attribute : "cfgInvoke", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure 4.10: Intermediate query for ROS component call graph.

```
// Retrieve each starting subscriber call edge of the intermediate
↪  inter-component call graph query
MATCH (:rosTopic)-[sub:subscribe]->(:rosSubscriber)
// Find all shortest paths consisting of one or more call edges starting with a
↪  subscribe edge
WITH *, apoc.cfgPath.rosFindPaths(sub, {
    relSeq : "call+",
    filter : "cFunction",
    config : [
        {name : "call", startLabel : "cFunction", endLabel : "cFunction",
        attribute : "cfgInvoke", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure 4.11: Suffix query for ROS component call graph.

```
// Find summary edge paths in descending order of number of components
MATCH path= (startPubTopic:rosTopic)-[:callflow*0..]->(endSubTopic:rosTopic)
RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure 4.12: System-wide query for ROS cross-component call graph.

Chapter 5

Query Analysis Evaluation of ROS-Based Software

In this chapter, we evaluate the precision improvements that result from the extensions to the extracted factbase model discussed in Chapter 3 and the query modifications to incorporate CFG-based checking presented in Chapter 4. Specifically, we examine the precision improvement against the performance overhead on multiple analyses over six different ROS-based software. The research questions that we are interested in are:

- R1: What are the precision improvements of performing analyses with CFG-based checking?
- R2: How much performance overhead is associated with CFG-based checking?

5.1 Selection of Subject ROS Systems

To evaluate the performance and precision of CFG-based checking, we have selected six ROS-based applications. This section provides an overview of the rationale behind the selection as well as the statistics for each of the extracted ROS systems.

5.1.1 Autonomous Vehicles

Autonomoose and WISE ADS are our primary subject systems because we were working with an automotive industrial partner and we wanted to demonstrate our work on some

reasonably-sized automotive software systems. Both Autonomoose and WISE ADS were developed at the University of Waterloo, hence, they were readily available to us.

Autonomoose

Autonomoose [27] is a self-driving vehicle project developed by the Waterloo Intelligent Systems Engineering (WISE) Lab at the University of Waterloo. The Autonomoose software comprises 18,738 lines of code across 15 components. Each software **component** is responsible for a specific goal or function in implementing autonomous driving. These components communicate with each other and with the automotive hardware components to achieve the general goal of transporting passengers safely and reliably [6].

WISE ADS

WISE Automated Driving System (ADS) [28] is derived from Autonomoose. The main difference between the two software systems is the programming language. The software components in Autonomoose are all written in C++, whereas WISE ADS components are implemented in a combination of C++ and Python3. Because Rex is designed extracts facts from only C/C++ source code, the evaluation of WISE ADS focuses solely on the C++ software components in WISE ADS.

Similar to Autonomoose, the software components in WISE ADS are mostly well-structured, where the source code files of the same component reside in the same file directory. Although there is one instance where two components are located in the same directory, the naming convention of the source code files can be used to differentiate each component's source code files with a regular expression during fact extraction.

5.1.2 Open Source Requirements

As stated previously, both Autonomoose and WISE ADS are proprietary, so we cannot share the dataset with other researchers. To evaluate our tools on software that could be shared, we considered the 179 publicly available ROS systems listed Kate et al.'s robotic system analyses paper [10]. Based on the constraints imposed by Rex and Neo4J discussed in the following subsections, eleven projects from Kate et al.'s paper satisfy all the criteria discussed. Due to the time constraint of setting up the extraction environment, we randomly selected three projects from Kate's paper. In addition, we included Autoware in our subject system list because it is an Autonomous Vehicle (AV) software [30] similar to

Autonomoose and WISE ADS, and is also open-source and listed as an important project in the ROS community [20].

Projects must use ROS1 (not ROS2)

ROS2 is the latest version of tools and libraries for developing software applications that use ROS-based communications. The communication constructs and related function calls in ROS2 are different from the ones used in ROS1. The main difference between the two is that ROS1 uses `advertise` and `subscribe` functions in the Node Handle object to create publishers and subscribers, whereas ROS2 uses `create_publisher` and `create_subscriber` in the Node object. Since Rex is built to analyze ROS1 projects, any projects that use ROS2 were removed from consideration. This criterion filters out 20 projects from those listed in S.Kate et al. [10].

Projects must have string-valued topic names

Another of Rex's limitations is its inability to extract facts about variable ROS topic names. Variable topic names are complex because they are prone to changes under different conditions, and their values may be affected by multiple variables. This thesis includes a small extension to Rex to handle cases of variable ROS topics with default string values, but this extension does not completely resolve the problem. We relax this project selection criterion to include projects that have some string-valued topics when creating publishers and subscribers, because otherwise there may not have been any acceptable projects of our studies. This criterion filtered out 3 additional projects from those listed in S.Kate et al. [10]. Despite our extension to Rex, our analyses may still miss data in which results pass through ROS communication with more complex variable topic names.

Projects must have subscribers and publishers that communicate with the same ROS topic

Our analyses perform cross-component queries but our pool of candidate subject systems includes projects that do not have ROS-based publishers and subscribers. Analyses of such projects would produce no results for many queries of interest, hence they are not worth analyzing. This criterion filters out another 101 projects.

Project must belong to an open repository

Kate et al.'s paper [10] was published in 2017 and some of the open-source projects referred to in that paper have become unavailable over time. Another 11 projects fall into this category and they are excluded from consideration.

Project must have four or more contributors

Many projects on Github are personal projects or class assignments and they are not meaningful to analyze because they typically have only one or two contributors. It is unlikely that we would find unexpected software behaviours caused by indirect component communication in these smaller projects. In addition, these projects often have only direct communications between software components, which is not very interesting to analyze. In the end, we considered only projects with four or more contributors, as these are more likely to be larger projects where indirect behaviours are more likely to occur. This criterion filters out 33 projects.

5.1.3 Statistics of the Subject Systems

The statistics of all the selected subject systems are summarized in Table 5.1. Table 5.2 provides a summary of the program facts extracted for each subject system.

Table 5.1: Open Source ROS Projects Statistics

Project Names	Contributors	LOC	# of C Files	# of Components
Autonomoose	24	18,738	156	15
WISE ADS	7	166,191	904	12
Autorally	21	9,885	79	14
Autoware	53	152,877	904	84
dyros_jet_dg	6	6,340	54	4
oko_slam	4	7,493	53	4

Table 5.2: Number of Facts Extracted From Open Source ROS Projects

Project Names	Non-CFG Entities	Non-CFG Relationships	CFG Entities	CFG Relationships	Total Entities	Total Relationships
Autonomoose	6,559	21,235	3,810	30,477	10,369	51,712
WISE ADS	8,495	26,878	4,674	38,998	13,169	65,876
Autorally	2,178	8,067	1,890	14,674	4,068	22,741
Autoware	43,313	150,119	32,969	269,631	76,282	419,750
dyros_jet_dg	5,480	13,560	4,681	30,095	10,161	43,655
oko_slam	1,424	3,401	1,515	9,311	2,939	12,712

5.2 Subject Analyses

The evaluation employed a variety of analysis queries of interest to software engineers including dataflow, control flow, and feature interaction type of analysis. Dataflow analysis track how data travels between variables, parameters, and function return statements in a program. Control-flow analysis report the possible flows of execution from one function or component to another. Feature interaction analysis are a combination of dataflow and control-flow analyses, and are used to detect unexpected behaviours in the software.

The analyses that we employed in our evaluation studies differ from the analyses that Muscedere employed in his evaluations in two important ways. First, Muscedere's queries mostly use control flow facts (i.e. *call* facts) as a proxy for dataflows; we revised such queries to use dataflow facts (i.e. *varWrite*, *parWrite*, and *retWrite* facts) instead because it produces a more precise result. Second, Muscedere's versions of behaviour and publisher alteration queries consider only the last component (starting with the receipt of a ROS message that leads to altered behaviours); we revised such queries to report the complete analysis result starting from the component that performs the triggering variable assignment through all intermediate components to the last component in which the altered behaviour resides.

Below are the ten analyses employed in our evaluation studies. The graphical pattern and query statements of each analysis can be found in Appendix C.

Inter-component-based communication analysis detects direct and indirect communications between two or more components through intracomponent dataflows and ROS inter-component communication.

Loop detection analysis reports instances of direct or indirect self-communications. This type of analysis is the same as inter-component-based communication analysis, except that it checks that the first and last components are the same.

Multiple publishers analysis detects instances when multiple publishers pass messages that lead to the same callback function in a ROS subscriber. Such instances identify publishers that might flood the event handler of a subscriber.

Race condition analysis detects when different publishers attempt to change the same variable in a subscriber component. This may cause nondeterministic behaviour if different orders of input messages lead to different system behaviours. Another potential problem is that the processing of later messages can overwrite the processing of previously received messages.

Behaviour alteration analysis detects potential unexpected behaviours due to the receipt of a message. Specifically, it checks whether the receipt of a ROS message can directly or indirectly influence the calling of a function in a control statement. This type of particular type of function call is seen as a potential behavioural change, initiated in another component that sent the ROS message.

Publisher alteration analysis detects instances where a received ROS message directly or indirectly causes a component to publish a message to a topic. The only difference between this analysis and the behaviour alteration analysis is whether the receipt of the ROS message leads to a potentially unexpected ROS publication or a potentially unexpected function call.

Intracomponent call graph analysis reports on paths of function calls that happen within the same component.

Inter-component call-graph analysis is a control-flow query that reports the paths of function calls across multiple components.

Recursion analysis is an intracomponent analysis that reports on paths of function calls that directly or indirectly call themselves in a program.

Variable triangle communication analysis is an intracomponent analysis that reports on variables that write to themselves after being used in a variable or parameter assignment to another variable.

5.3 ROS System Experiments

To answer the two research questions listed at the beginning of this chapter, we performed ten analyses across the six ROS systems. For each of the analyses, we ran two sets of queries to evaluate the precision and performance of our extensions: (1) a **CFG-based** analysis that uses versions of the subject queries that perform CFG-based checking, and

(2) a **baseline** analysis that uses versions of the subject queries that do not perform CFG-based checking (and thus do not check whether analysis results have corresponding execution paths).

All of these analysis queries were run on an Ubuntu 18.04.6 machine with an Intel(R) Xeon(R) processor with 12 CPUs.The query analyses are performed with a Neo4J database, using a single CPU core, a heap size of 100 GB of the heap size, a page cache size of 100 GB, and access to a total disk space of 459 GB.

5.3.1 Precision Evaluation

Due to time constraints, an in-depth precision evaluation is performed on Autonomoose and WISE ADS. These were the only systems to which we had direct access to the development team. Muscedere et al. reported that their suffix intercomponent behaviour-alteration analysis of Autonomoose exhibits a low precision of 8.9% [14]. The precision of a system-wide behaviour alteration analysis would be worse. CFG-based checking is devised to improve the analysis precision by checking that each analysis result has a corresponding feasible execution path. Research question RQ1 evaluates the precision of analyses employing CFG-based checking by comparing the precision of a CFG-based behaviour-alteration analysis against a baseline behaviour-alteration analysis, as applied to Autonomoose software. For better comparison, both the CFG-based and baseline analyses were conducted using two-phased queries; the CFG-checked queries employ on-the-fly checking.

Autonomoose

The baseline analysis reported a total of 350,985 behaviour alteration results and the CFG-based analysis reported 5,872 results. However, we assume that behaviour alterations involving only two components represent expected behaviour and are not interesting results. Thus, we filtered out the two-component results, leaving 221,708 baseline results versus 502 CFG-checked results. In this study, CFG-based analysis results in a 99.8% reduction in the number of results for engineers to investigate (see Table 5.3).

We examined the precision of the reported behaviour-alteration results using a combination of manual inspection and Linux command-line tools. Table 5.3 shows the results of our examination. All of the CFG-checked results reported by the baseline analysis, plus an additional 221,206 results unique to the baseline analysis.

Consider first the 221,206 results that are unique to the baseline analysis. Recall in Section 4.2.3, the analysis results were constructed by combining the subquery results

Table 5.3: Autonomoose 3+ Components

	Total Paths	**Valid Paths**	**Invalid Paths**	**Source of Imprecision**
Common	502	156	346	- Assignments in the same CFG block - Return node associated with all calling contexts
Only in CFG*	0	0	0	
Only in NCFG†	221,206	0	221,206	- Infeasible execution path - Return node associated with all calling contexts - Multiple assignments to the same variable to the same CFG block
Total CFG*	502			
Total NCFG†	221,708			

* CFG in the table represents results from CFG-checked analysis
† NCFG in the table represents results from baseline analysis

from phase 1 and phase 2 analyses. By taking advantage of the fact that many analysis results have common subresults, finding an invalid subpath in an intracomponent dataflow subresult would invalidate all full paths that include that dataflow. In general, eight unique intermediate intracomponent dataflow subresults are in these 221,206 baseline analysis results. Considering these eight common intracomponent dataflow subresults:

- One of the intracomponent dataflow subresults is invalid because it includes an invalid function return to the wrong calling context. It is impossible with static analysis to disambiguate the calling context of a function call, as described in Section 4.2.2. As a result, to be conservative in our analyses, the factbase retains relationships from each return node to all possible calling contexts. Thus, a dataflow path can be invalid if the path flows into a function from one calling context but returns to a different calling context. There are 123,737 baseline paths that contain this invalid function return.
- Six of the eight intracomponent dataflow subresults are invalid due to an infeasible execution path of two or more consecutive dataflow assignments. In other words, they failed the CFG-checked or intervening check described in Section 4.1. These seven infeasible intracomponent dataflows affect another 97,463 baseline results.
- One of the intracomponent dataflow subresults had multiple assignments to the same variable in the same CFG block problem. The order of execution of multiple dataflow relationships that occur in the same CFG block is unknown. Thus, this analysis will retain the results that reflect any of these same-CFG-block dataflows even though only one is correct with respect to the source code.

Eight invalid intracomponent subpaths invalidated all 221,206 paths returned by the NFG-checked analysis that were not returned by the CFG-based analysis.

Finally, of the 502 results that are common to both analyses, 156 are valid and 346 are invalid. The 346 invalid results are due to (1) function return nodes that are associated with all of their respective function's calling contexts as described above, and (2) multiple assignments to the same variable in the same CFG block. The precision of behaviour-alteration analysis improves to 31.1% with CFG-based checking.

WISE ADS

The behaviour alteration analysis of WISE ADS reports a total of 5,635 baseline results and 32 CFG-checked results. When considering results that correspond to three or more components, we are left with 468 baseline results and no CFG-checked results.

We examined the baseline results, and found that they are all invalid for the same reasons and sources of imprecision mentioned in the Autonomoose case study (discussed in Section 5.3.1). The results of our precision analysis for the behaviour-alteration evaluation involving three or more components in WISE ADS are presented in Table 5.4.

Table 5.4: WISE ADS 3+ Components

	Total Paths	**Valid Paths**	**Invalid Paths**	**Source of Imprecision**
Only in CFG*	0	0	0	
Only in NCFG†	468	0	468	- Infeasible execution path - Return node associated with all calling contexts - Multiple assignments to the same variable to the same CFG block

* CFG in the table represents results from CFG-checked analysis
† NCFG in the table represents results from baseline analysis

5.3.2 Performance Evaluation

Research question RQ2 assesses the performance overhead that is incurred in order to achieve this increased level of precision. As described in Section 4.2, we propose two-phased query processing and on-the-fly checking to mitigate against expected overhead problems.

The performance evaluation is divided into two sub-categories:

1. **Setup** time includes the total time Rex takes to extract facts, and total time of importing component-based facts into Neo4J for phase 1 and full system facts for phase 2 analysis for each software system.

2. **Analysis** time includes the time to run the subqueries in both phase 1 (component-based queries) and phase 2 (system-wide queries), and the time it takes to combine the results from these two phases.

Tables 5.5 (a)-(j) show the reduction in the number of results due to CFG-based checking and performance overhead for each analysis. The performance reported for each software system is the sum of the execution times for fact extraction, fact import action, query processing, and the combining of phase 1 and phase 2 results. The multiple-publishers analysis results displayed in Table 5.5 (c) show only the baseline results without CFG checking because this analysis involves only cross-component facts that have no corresponding CFG paths that need to be checked.

As results in the performance evaluation tables show the CFG-based checking reduces the number of reported analysis results compared to the non-CFG-based analysis. Fortunately, CFG-based checking does not exhibit overheads in some analyses. The degree of performance overhead of CFG checking depends on the analysis and the size of the program being analyzed.

Autoware is the largest software system that we performed analysis on, hence most of its CFG-checking analyses showed a positive overhead, except for the intracomponent call-graph analysis. One reason this analysis may have negative overheads is that calls to library functions and functions with no definitions are filtered out in the early stages of the CFG-based checking because the entry blocks of these functions do not exist, thus causing the CFG-based test to fail.

Analyses with variable dataflows such as race conditions, behaviour alteration, publisher alteration, and variable triangle communication typically produce higher positive overheads compared to other analyses. We hypothesize that these overheads are due to longer path results which require additional CFG checking when determining the CFG feasibility. The amount of dataflow facts is always greater than control flow facts, hence causing additional endpoints to consider and additional consecutive facts to check for.

Although there are some large overheads with CFG checking of transitive dataflows, the increase in precision still outweighs the overhead because there are significant improvements in the filtering of paths, greatly reducing the manual examination of results to be performed by the engineers.

Table 5.5: Performance Evaluation of Query Analysis (1)

(a) Inter-Component-Based Communication (Two-Phased)

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	15	4	-73.3%	585,772.49	23,632.47	581,673.81	26,591.45	-0.2%
WISE ADS	12	0	-100.0%	826,425.75	20,969.98	831,134.07	23,554.98	0.9%
Autorally	3	1	-66.7%	532,603.59	20,702.58	536,591.48	21,619.06	0.9%
Autoware	1,281	249	-80.6%	3,862,232.07	126,859.93	4,195,526.70	185,894.13	9.8%
dyros_jet_dg	0	0	-	665,747.00	6,789.19	677,748.10	9,360.06	2.2%
oko_slam	0	0	-	212,407.13	7,372.81	206,309.79	7,461.27	-2.7%

(b) Loop Detection (Two Phased)

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	16	15	-6.3%	584,445.02	24,841.20	587,141.70	27,774.40	0.9%
WISE ADS	3	2	-33.3%	827,433.64	20,658.48	831,977.80	24,520.14	1.0%
Autorally	6	4	-33.3%	531,070.83	20,745.94	535,172.49	21,633.68	0.9%
Autoware	85	46	-45.9%	3,858,219.98	126,611.60	4,188,156.68	181,892.97	9.7%
dyros_jet_dg	0	0	-	667,314.22	6,589.75	681,938.33	7,680.31	2.3%
oko_slam	0	0	-	215,258.92	7,209.85	210,296.79	7,512.09	-2.1%

(c) Multiple Publishers

	Number of Paths	Performance (ms)	
		Setup	Analysis
Autonomoose	26	27,524.08	2,809.47
WISE ADS	8	123,997.18	1,407.03
Autorally	22	23,542.34	1,501.36
Autoware	368	210,361.46	2,125.75
dyros_jet_dg	0	24,561.22	1,536.24
oko_slam	0	21,449.23	1,413.15

* CFG in the table represents results from CFG-checked analysis
† NCFG in the table represents results from baseline analysis

Table 5.5: Performance Evaluation of Query Analysis (2)

(d) Race Conditions (Two Phased)

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	2,815	185	-93.4%	763,074.86	338,238.54	599,462.36	198,685.99	-27.6%
WISE ADS	521	15	-97.1%	904,375.99	96,428.99	852,869.66	125,569.99	-2.2%
Autorally	16	0	-	536,582.03	26,222.66	536,203.36	48,895.69	4.0%
Autoware	6,303	1,351	-78.6%	4,906,968.73	16,653,576.90	5,634,874.78	430,230,480.44	1921.6%
dyros_jet_dg	0	0	-	676,835.66	10,591.02	687,881.44	190,510.76	27.8%
oko_slam	0	0	-	218,114.00	6,239.39	208,690.31	6,730.63	-4.0%

(e) Behaviour Alteration (Two-Phased)

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	221,708	502	-99.8%	583,290.76	228,271.53	580,946.07	432,601.98	24.5%
WISE ADS	468	0	-100.0%	831,667.62	91,583.84	835,881.39	331,495.82	26.4%
Autorally	4,330	0	-100.0%	529,384.23	31,174.95	535,131.06	72,246.86	8.4%
Autoware	118,913	2,553	-97.9%	3,856,460.00	9,126,885.37	4,189,099.86	297,093,079.64	2,220.5%
dyros_jet_dg	0	0	-	666,366.06	13,179.43	680,674.60	914,710.60	134.8%
oko_slam	0	0	-	214,517.34	8,142.44	210,175.60	10,995.62	-0.7%

(f) Publisher Alteration (Two-Phased)

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	2,534	0	-100.0%	581,584.77	90,326.29	579,222.74	374,049.52	41.9%
WISE ADS	0	0	-	828,942.32	78,741.15	833,252.51	268,279.98	21.4%
Autorally	0	0	-	530,988.02	31,507.86	536,050.06	39,229.25	2.3%
Autoware	4,578	0	-100.0%	3,905,987.13	1,172,544.68	4,227,421.57	51,497,652.61	997.3%
dyros_jet_dg	0	0	-	665,738.03	12,419.96	678,605.14	622,060.08	91.8%
oko_slam	0	0	-	215,496.61	9,022.86	205,735.71	10,487.58	-3.7%

* CFG in the table represents results from CFG-checked analysis

† NCFG in the table represents results from baseline analysis

Table 5.5: Performance Evaluation of Query Analysis (3)

(g) Intracomponent Call Graph

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	9,171	3,946	-57.0%	28,188.94	6,750.91	30,515.54	3,852.51	-1.6%
WISE ADS	15,563	5,492	-64.7%	124,253.24	10,747.94	126,899.59	4,796.61	-2.5%
Autorally	530	53	-90.0%	23,659.85	2,816.19	27,098.38	2,340.64	11.2%
Autoware	165,709	94,173	-43.2%	210,317.27	172,339.25	204,558.46	127,428.82	-13.2%
dyros_jet_dg	8,752	3,272	-62.6%	24,624.16	7,104.70	29,192.10	3,825.97	4.1%
oko_slam	1,202	595	-50.5%	21,467.43	2,103.10	22,824.72	1,840.25	4.6%

(h) Inter-component Call Graph (Two-Phased)

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	8,913	1,283	-85.6%	578,453.72	29,190.49	580,673.84	31,879.10	0.8%
WISE ADS	774	83	-89.3%	825,029.01	26,828.92	834,302.36	30,804.46	1.6%
Autorally	2,887	1,481	-48.7%	529,325.97	25,139.64	536,199.14	26,258.58	1.4%
Autoware	40,928	6,300	-84.6%	3,806,172.01	165,946.70	4,168,761.62	206,644.59	10.2%
dyros_jet_dg	8	2	-75.0%	665,283.93	10,006.58	679,430.92	13,426.01	2.6%
oko_slam	6	1	-83.3%	215,235.98	8,173.90	204,874.65	8,413.49	-4.5%

(i) Recursion

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	0	0	0.0%	28,314.42	3,022.98	30,241.89	2,106.24	3.2%
WISE ADS	3	3	0.0%	124,390.58	1,563.63	127,300.62	354.33	1.4%
Autorally	1	1	0.0%	23,623.68	1,579.08	26,986.16	1,863.35	14.5%
Autoware	27	27	0.0%	210,683.27	2,298.63	203,526.42	15,820.73	3.0%
dyros_jet_dg	0	0	-	24,501.37	1,498.15	29,127.36	2,303.78	20.9%
oko_slam	1	1	0.0%	21,478.37	1,432.01	22,500.89	1,509.70	4.8%

* CFG in the table represents results from CFG-checked analysis

† NCFG in the table represents results from baseline analysis

Table 5.5: Performance Evaluation of Query Analysis (4)

(j) Variable Triangle Communication

	Number of Paths			Performance (ms)				
	NCFG†	CFG*	% Filtered	NCFG†		CFG*		% Overhead
				Setup	Analysis	Setup	Analysis	
Autonomoose	102	78	-23.5%	28,745.01	1,704.23	30,724.33	4,973.40	17.2%
WISE ADS	129	110	-14.7%	123,934.50	1,879.05	127,222.85	4,942.48	5.1%
Autorally	54	40	-25.9%	23,572.15	1,408.78	26,471.70	2,756.19	17.0%
Autoware	1,682	1,260	-25.1%	210,013.64	2,766.09	203,848.96	53,962.32	21.2%
dyros_jet_dg	75	49	-34.7%	30,027.63	1,848.86	28,890.60	14,370.11	35.7%
oko_slam	8	6	-25.0%	23,653.92	1,619.25	23,071.70	2,246.82	0.2%

* CFG in the table represents results from CFG-checked analysis

† NCFG in the table represents results from baseline analysis

Chapter 6

Neo4J Queries with CFG-Based Checking on Non-ROS Software

ROS applications use constructs in the ROS framework to perform cross-component communications. In non-ROS applications, such communications are often achieved by cross-component function calls. This chapter discusses the modifications made to a factbase and to the behaviour alteration query of Chapter 4 due to resource constraintsto run these analyses on non-ROS software.

6.1 Two-Phased Queries (Non-ROS)

Non-ROS applications typically use parameter passes and function returns to communicate between components. These constructs may not have well-defined entry and exit points for each component. Furthermore, the code associated with each component may be unclear compared to ROS programs, where each component has its own directory. Therefore, some additional pre-processing is required before performing a two-phased analysis.

First, we use Rex to create the entire factbase without considering components. Second, we partition the full factbase into component factbases using a user-provided mapping of source code files to identified components. Each entity fact is assigned to the component factbase associated with the source-code file containing the entity's definition. When assigning a relationship facts to a component factbase, the locations of both the fact's source and destination endpoint entities need to be considered. A **same-component** fact is a relationship fact whose source and destination entities belong to the same component.

Such facts can simply be added to their endpoint entities' component factbase. In contrast, a **cross-component** fact requires special attention as it defines entry and exit points for the affected components. For each cross-component fact with the relationship type of *varWrite*, *parWrite*, *retWrite*, or *call*, we create both a *cCompEnd* and a *cCompStart* entity fact to represent exit and entry points, respectively, for their components. Along with these new entity facts, we create a relationship fact that relates the *cCompEnd* entity with the corresponding *cCompStart* entity. We also create relationships between the *cCompStart*/*cCompEnd* entities and their corresponding cross-component program-element facts. Figure 6.1 shows the additional nodes and edges, coloured in blue, that are created for each type of cross-component fact.

Consider the cross-component *call* fact in Figure 6.1 as an example. Suppose function `f1` is in component **c1**, function `f2` is in component **c2**, and **c1** does not equal to **c2**. This cross-component function *call* fact will trigger the creation of four nodes and six edges to facilitate the intra- and inter-component analyses. A *cCompStart* and a *cCompEnd* node will be created for both the calling (`f1`) and called (`f2`) function. The path from `f1` to the compEnd of `f1` , to the compStart of `f2` , to the function `f2` models the switching of context from caller to callee. The path from `f2` to the compEnd of `f2` , to the compStart of `f1` , to the function `f1` models the returning from the called context. If this function call passes parameters, then additional pre-processing would be required. The *cCompStart* and *cCompEnd* nodes for a cross-component *parWrite* would be the same as for the cross-component *call*. The formal parameter would be linked to the *cCompEnd* node and the actual parameter would be associated with the corresponding *cCompStart* node. Similarly, if the function call occurred in a variable assignment, we would use the same *cCompEnd* and *cCompStart* nodes for the corresponding cross-component function return (*retWrite*).

In addition to the above changes to the factbase, two revisions are made to the analyses. First, we revise the intracomponent subqueries to use the new entity facts *cCompStart* and *cCompEnd* (rather than ROS publish/subscribe commands) as the components' entry and exit points. Second, we revise the system-wide query to refer to the new *compCall*, *compReturn*, and *compWrite* inter-component facts. Everything else concerning combining the subqueries and the summary results remains the same.

6.2 Neo4J Queries and APOC

We extended the APOC library with a new function to perform CFG-based checking on non-ROS-based systems. This function returns a CFG-checked path meeting all requirements and options specified by the user. The only difference between this function and

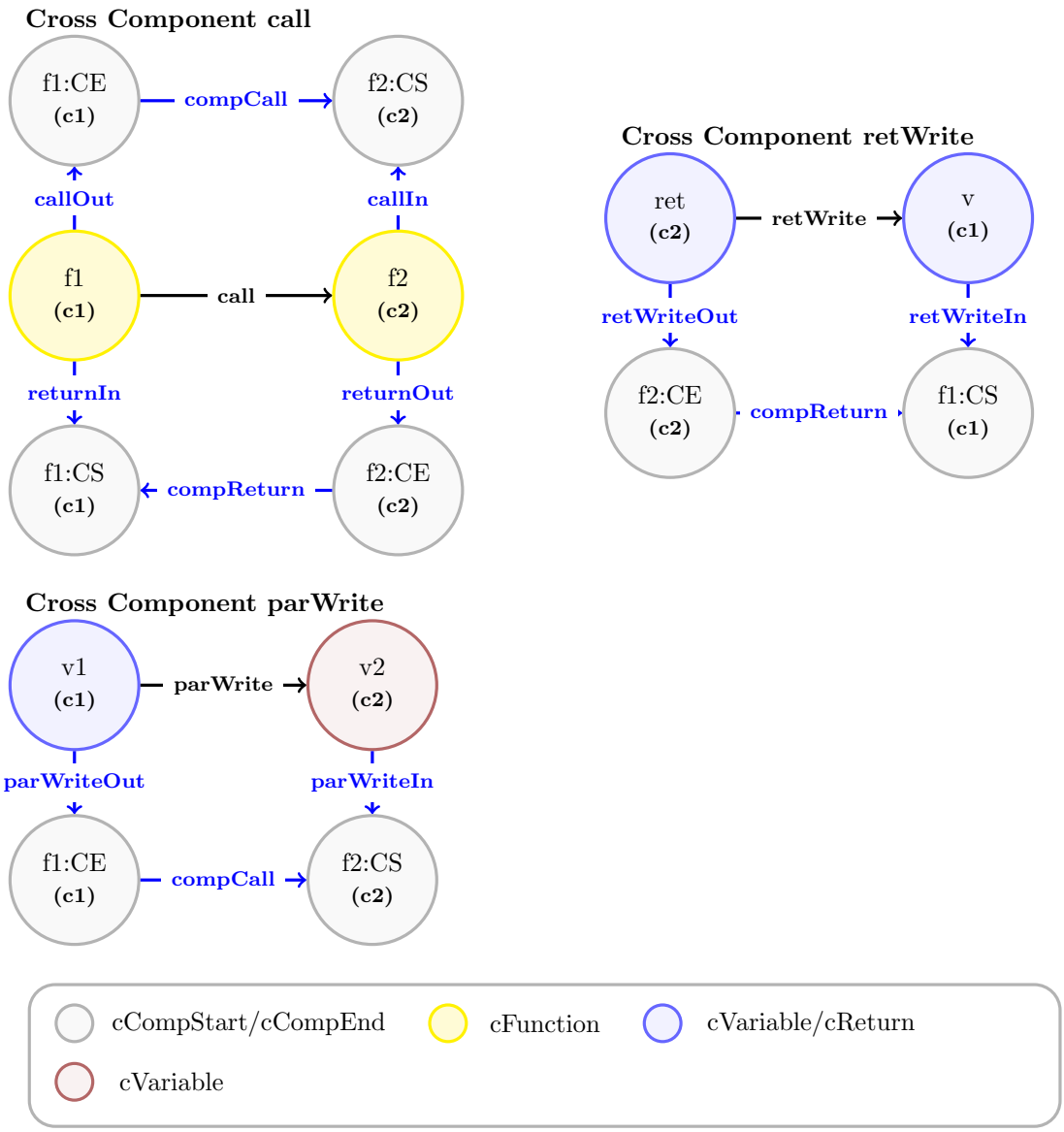


Figure 6.1: New nodes and relationships for cross-component facts.

the one described in Chapter 4 is that it does not perform CFG checks on the component endpoint nodes and edges.

apoc.cfgPath.NonROSPath(start, {endE, endN, cfg, relSeq, repeat, shortest, config, filter})

- *start*: a starting graph element (can be a node or an edge); cannot be null
- *endE*: an ending edge. The default value is null
- *endN*: an ending node. The default value is null
- *cfg*: a boolean value indicating whether or not to perform CFG-based checking. Default value is *true*
- *relSeq*: the pattern of edges that define the analysis query
- *repeat*: whether or not the *relSeq* identified above is allowed to be repeated (as when performing transitive closure). Default value is *false*
- *shortest*: a boolean value indicating whether or not the analysis uses our shortest path function (described in Section 4.2)
- *config*: a dictionary that describes each edge type in *relSeq*, the pattern of *nextCFG-Block* edges connecting the relationship's source and destination CFG nodes
- *filter*: the types of CFG nodes allowed in the CFG-checked analysis results

A two-phased behaviour-alteration query for a non-ROS-based software system is depicted in Figure 6.2. The start and end points of these subqueries are *cCompStart* and *cCompEnd* nodes and their respective edges. All other parts of the subqueries remain the same as the ROS-based queries depicted in Figure C.6.

As discussed previously, Neo4J Cypher without APOC does not support transitive closure over complex patterns. This affects our system-wide phase 2 query which looks for (1) a prefix path, (2) one or more repetitions of a *compCall*/*compReturn*/*compWrite* relationship followed by a dataflow relationship, and (3) an ending suffix path. We experimented with a custom Neo4J function to support a transitive closure of the second subpattern, but it was determined to be too slow. Instead, we pose a more general Cypher query and filter the results using a Python script to remove phase-2 results that violate the query's pattern.

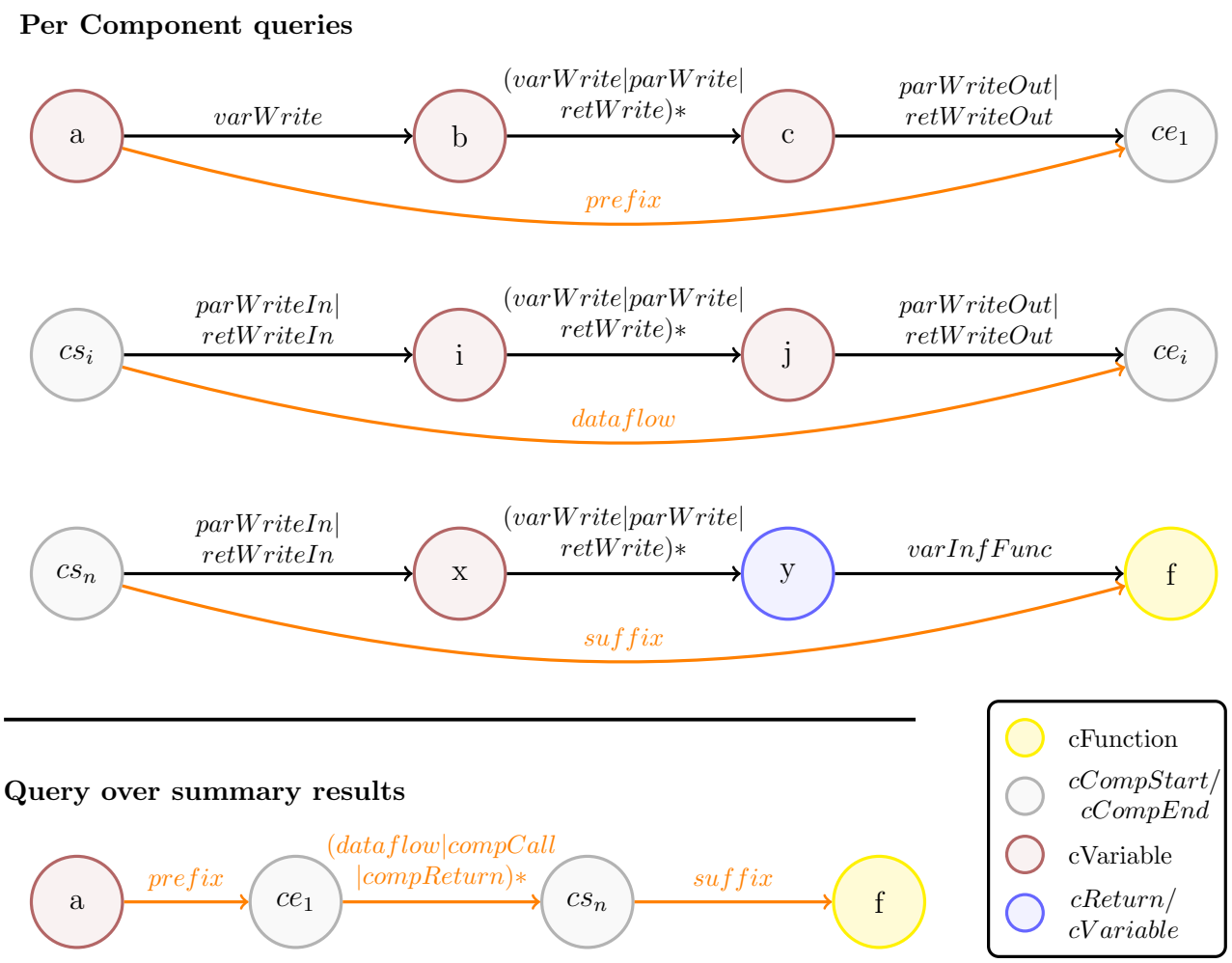


Figure 6.2: The intracomponent and system-wide behaviour-alteration queries for non-ROS applications

Chapter 7

Query Analysis Evaluation of Non-ROS Software

Non-ROS-based applications do not have well-defined entry and exit points in each of their components. To enable two-phase analysis in the absence of ROS component delimiters, we used the component start and endpoint nodes and corresponding edges that were introduced in the last chapter. To evaluate the scalability of CFG-checked analyses on non-ROS-based software, we applied such analyses to an industrial automotive controller (hereafter called the Controller). The Controller contains 1,639,822 lines of code with 1,356 components (more details shown in Table 7.1). The Controller is much larger than the subject systems in Chapter 5. Therefore, we use the Controller in the scalability experiment, to understand whether our analyses can handle large industrial software systems. This chapter describes only an evaluation of the performance of behaviour alteration analysis on the Controller due to resource constraints. Similar to the performance evaluation discussed in Chapter 5, the evaluation is performed by comparing the performance of a baseline analysis without CFG checking to the with CFG checking. A precision analysis was not conducted because we did not have direct access to the source code, just to the extracted factbase.

7.1 Pre-Processing

As discussed in the last chapter, non-ROS software applications do not have clear delimiters for components like ROS applications. A component is not a well-defined software unit in

Table 7.1: The Controller Statistics

The Controller Statistics	
Lines of Code	1,639,822
# of C Files	8,458
Non-CFG Entities	411,959
Non-CFG Relationships	1,065,447
CFG Entities	699,877
CFG Relationships	3,666,710
Total Entities	**1,111,846**
Total Relationships	**4,732,157**

C/C++, and in some cases, all source code files are stored under a single file directory. As such, a filename-to-component mapping file must be provided as a distinct input to the pre-processing script that partitions the full system factbase into component factbases for intracomponent analyses. This mapping file is a comma-delimited file with the component name first, followed by the file name. This pre-processing script along with the mapping file can be applied to any non-ROS projects that (1) partitions the Rex-extracted factbase into component factbases, (2) creates the component endpoint nodes to model the entry and exit points of each component, and (3) appends the component endpoint nodes and cross-component *compCall/compWrite/compReturn* edges to the full system factbase.

In our behaviour alteration analysis of the Controller, the phase 2 path filtering script described in Section 6.2 reduces the CFG-checked phase-2 analysis results from 229,107 to 53,813 paths and reduces the baseline phase-2 analysis results from 255,757 to 65,115 paths.

The intermediate results from phase-1 and phase-2 analyses succeeded with minimum issues. However, when combining these sub-results, we exceeded the available space on our server. The final behaviour-alteration baseline analysis results consumed a total space of 1230.40GB. Our solution to this problem of space constraints was to process the analysis results in small batches. Specifically, we processed the baseline phase-2 analysis results 25 paths at a time. For each batch of 25 results, we ran our post-processing scripts to remove invalid results before processing the next batch of query results. In comparison, the CFG-checked analysis results are much smaller and required 2.9GB of storage.

7.2 Performance Analysis

As stated previously, the number of behaviour-alteration results for the Controller is extremely large in comparison to the number of results for Autonomoose. We do not have access to the source code to evaluate the precision of the analysis by comparing the report paths to the source code. As such, we evaluated only the performance of the analyses and compared the performance of a CFG-checked analysis against the performance of a baseline analysis. We used two-phased query processing and on-the-fly checking, in combination with the post-processing steps described above, to perform a behaviour alteration analysis of the full Controller. The total number of behaviour alteration instances that are CFG-verified is 1,092,919 and the number of behaviour alteration results when there is no CFG-based checking is 307,273,520. Although 1,092,919 is a lot of results to analyze, the results are reported in order of their length where the length of the result is the number of components involved. Thus the engineers can examine the behaviour alterations with the longest dataflows first, given that long dataflows are likely to be the most surprising and hardest to detect, as shorter ones are likely intended communication. Table 7.2 shows the distribution of the lengths of the 1,092,919 verified behaviour alterations reported for the Controller.

Table 7.2: Length of CFG-Verified Behaviour Alterations

Number of Components	**Number of Behaviour Alterations**
3	290,191
4	639,108
5	47,219
6	113,723
7	2,678
Total	1,092,919

As shown in Table 7.3, it took a total of 51,433,069.47ms (14hrs 17mins 13s 69ms) to perform the analysis without CFG-checking and 1,064,619,711.11ms (12 days 7hrs 43mins 39s 711ms) to perform the CFG-checked analysis. The CFG-based checking resulted in a 99.6% reduction in reported results. Similar to the performance results in Autnomoose, the phase-2 queries are faster than phase-1 queries because there are fewer paths traverse. In addition, the phase-2 queries and post-processing are faster with CFG-based checking than without it because many invalid paths are filtered out in phase 1. As expected, the

improvement in precision incurs a performance cost. In this case, the cost is an overhead of 1,969.9%. Because the phase-1 analyses are component-based and are performed on distinct component factbases, they could be performed concurrently. Aggressive concurrent processing of the phase-1 analyses in which all intracomponent subqueries run in parallel, would be bounded by the time it takes to perform the most expensive subqueries on the most expensive component. In our behaviour alteration analysis of the Controller, the largest time to perform a single intracomponent is approximately 8 days. Thus the lower bound on CFG-checked behaviour alteration analysis of the Controller is about 8 days, which has a performance overhead of 53.6% over a concurrent behaviour analysis, without CFG checking of the controller.

Moreover, when the software evolves, the phase-1 results of unchanged components are preserved and only the changed components need to be re-analyzed as well as the phase-2 query and post-processing. Thus, subsequent analyses would likely be much faster.

Table 7.3: GM Controller - Performance (ms)

	Number of Path			**Performance (ms)**		
	NCFG	**CFG**	**% Difference**	**NCFG**	**CFG**	**% Overhead**
Phase 1				51,246,117.15	1,064,513,971	1977.3%
Phase 2				123,302.27	105,500.14	-14.4%
Path Filtering				367.38	227.21	-38.2%
Batch Processing				63,282.67	12.32	-100.0%
Final Behaviour Alteration Path	307,273,520	1,092,919	-99.6%	51,433,069.47	1,064,619,711.11	1969.9%

Chapter 8

Sources of Imprecision

The motivation behind our work is to address sources of imprecision in Muscedere's analyses of software models by enriching our factbase with control-flow facts and revising our analysis queries to consider these new facts. This chapter discusses the different sources of imprecision and the degree to which each has been resolved.

8.1 Resolved

The key source of imprecision in previous work was parameter-passing dataflows and this imprecision is completely resolved. In our previous work, the factbase did not include parameter assignments, and all program variables were treated as potential actual parameters to any function with formal parameters, resulting in a high percentage of false positive query results. To resolve this problem, a new type of fact that records parameter assignments is created and this allows the analyses to reduce invalid function-call-related dataflows by considering only the flows from actual parameters to formal parameters. Infeasible function-related dataflows via parameterless function calls are also avoided.

8.2 Improved

Five sources of imprecision are greatly improved but not completely addressed. The first such imprecision involves dataflows through function return values, which our previous work did not track at all; only flows into function calls were previously tracked. The

extension of the factbase to include function-return facts improves the completeness of our dataflow results. Function-return dataflows are restricted to variables that appear in a return statement. However, we cannot achieve 100% precision because we cannot always track a function-return dataflow to its corresponding calling context. Instead, we associate a function return with *all* its possible calling contexts to ensure that the correct function-return dataflow is included in the analysis results; but including function-return dataflow to *all* calling contexts introduces a new source of imprecision. One way of reducing this imprecision in our queries is to create a function-call stack in the path-searching algorithm and filter out dataflow paths that violate this stack. This could be explored in future work.

A second source of imprecision relates to the order of execution of facts in a result path, which our CFG-based checking is designed to address. Specifically, the CFG-based checking associates each fact with its CFG blocks, and then it checks that paths of consecutive facts in an analysis result have corresponding CFG paths among the consecutive facts' CFG blocks, as a proxy for ensuring that reported analysis results correspond to possible execution paths in the program. Analysis with CFG-based checking eliminates 99.6% of the unchecked behaviour alteration results detected in the software controller analyzed in Chapter 7. Unfortunately, CFG-based checking does not resolve all imprecisions related to feasible execution paths because when two variable assignments occur in the same CFG block, it is impossible to differentiate the respective order of these assignments. In such cases, the analysis results will include both assignment orderings to ensure that the actual assignment ordering is included in the analysis results, but it leaves a residual source of imprecision.

A third source of imprecision is that software entities of the same name can be declared in different files. This imprecision is most prevalent in non-object-oriented C code (e.g. the software controller analyzed in Chapter 7) because there are no classes to use as another layer in the node identifiers. We have modified Rex to use an entity's location to differentiate between same-named distinct entities, by appending file names to node identifiers. The modified Rex also adds a new attribute to each software entity to indicate the location of the entity's declaration. Furthermore, in our pre-processing script that assigns software entities and facts to corresponding components, we associate a software entity with the component that contains its definition rather than the components that contain its declarations. The result is an improved model of the program because many function entities are defined and declared in different components to facilitate cross-component communication through function calls.

A fourth source of imprecision concerns static functions with multiple definitions. We create separate software entities and relationships for each static function definition. The scope of a static function is restricted to the file in which it is defined. We have revised

Rex to differentiate each of these static function definitions by adding to the identifier of a static function the name of the file containing the definition. This revision improves our factbase because many of these functions have different bodies, different internal facts, and different function callers. A second positive impact of this Rex extension is that the imprecision described above, of function-return dataflows, is reduced if static functions have different callers, and thus the returned dataflows can be narrowed to the callers of that static function.

Even though our new work on CFG-based checking inherits any limitations of Clang's control flow graph, it includes a lack of support for try-and-catch statements. This source of imprecision is also a static-analysis limitation because it is hard to identify the causes of exceptions statically. We could map each CFG block in each try statement to every entry CFG block of every related catch statement, but this may cause an explosion in paths because the exception may be triggered by some function called inside the try block.

8.3 Lingering

Many sources of imprecision are due to unsupported C/C++ language features, such as variable pointers or references, polymorphic function calls, or internal fields of objects in a C++ program. The former imprecision is due to the inability to track runtime values of pointers in a static analysis. The latter two sources of imprecision are of lower priority because the program is written in C. We have no plans to support these language features in the near future.

Chapter 9

Related Work

Several researchers use queries in a graph database to analyze source code programs. Typically, these works build software models of facts from the source code, store them in graph databases, and then express analyses as database queries [21]. For example, Wiggle [29] traverses through the Java AST and converts these AST elements to graphical elements in Neo4J before making queries. Frappé [8] uses wrapper scripts and a modified Clang to extract information from the source code and it uses Neo4J to query and visualize C/C++ dependencies. However, none of these source code queries consider information about execution paths.

In addition, some research analyses were performed on ROS-based applications. Witte et al. [32] use static analysis on the launch files and dynamic analysis on the ROS communication nodes to ensure the consistency of the architecture of the overall program. Santos et al. [23] create a metamodel of the overall architecture of the ROS system to perform analyses such as the ROS computation graph. These research works analyzed ROS-based systems at the component level, instead of considering control flows and dataflows that occur within each of these components.

Datalog can be an alternative database used to analyze graphical software models. Much research has been conducted using Datalog for static program analyses [7, 24]. Whaley et al. use binary decision diagrams extracted from Java and C programs and Datalog to perform different point-to analyses [31]. DOOP [26] is another framework developed using Datalog to perform these point-to-analyses on Java programs. However, a concern with using the analyses performed on Datalog is that they cannot track the complete paths of these analyses. They only return the endpoints indicating that the start point is related in some ways to the end point. This can be a challenge for engineers to pinpoint the potential

problem in the source code.

Chapter 10

Conclusion

In this thesis, we present extensions to Rex including refined function-call-related dataflows and CFG facts, and revisions to Neo4J queries of ten different analyses to perform CFG-based checking. A performance and precision evaluation of the behaviour-alteration analysis was performed on Autonomoose [27] and WISE ADS [28]. The CFG-based checking filters out 99.8% of the results with an overhead of 29.4% on Autonomoose and filters out 99.5% of the results with an overhead of 26.6% on WISE ADS. In addition, we employed the CFG-based analysis on four other open-source ROS-based software and a non-ROS-based software, the Controller.

10.1 Limitations

Limitations of our work are discussed in detail in the soundness chapter of this thesis. They can be generally divided into three categories: (1) Clang CFG limitations, (2) unsupported C/C++ features, and (3) other static analysis limitations.

Clang CFG limitations include the inability to differentiate the order of execution of facts in the same CFG block. Thus, when dataflow facts belong to the same CFG block, the CFG-based checking is unable to filter out invalid intra-block dataflow paths, leaving a residual imprecision of analysis results that do not correspond to valid execution paths. A second limitation under this category relates to the fact that Clang CFG cannot determine the execution paths of try-and-catch blocks.

Unsupported C/C++ features include the inability to track changes made to variable pointers or references, polymorphic function calls, and internal fields of objects. In the

former, it is not possible to know which program entities are referenced when the value of a pointer or reference is dereferenced, because in static analysis, variable runtime values are unknown. The latter feature is a problem because Rex records only changes to a parent object, not changes to an object's member variables.

Other static analysis limitations include the dataflow from a function return to the correct caller. To resolve this issue, our factbase associates a function return with all possible callers of the function. This will cause the analysis results to include invalid paths from (1) a calling function to (2) a called function, which returns control to (3) another function that is not the caller. This limitation creates false positives that associate a call with the wrong caller but it will also include the true positive in the final results.

10.2 Future Work

This thesis suggests several areas of future work: (1) further improvements to the precision of factbase analyses, (2) improvement to the performance of CFG-based checking, (3) a function-call stack to enhance the precision of dataflows from function returns to their calls, (4) integration to support ROS2.

One approach to improving the precision of factbase analyses is to extend Rex and CFG facts with symbolic-execution conditions. **Symbolic execution** [3] determines the inputs or conditions that cause a program fragment to execute. By embedding symbolic execution conditions onto CFG edges, we can identify infeasible execution paths and perform additional filtering on those results.

A second area of future work would be to improve the performance of CFG-based checking. It took over 87 hours to perform the full behaviour-alteration analysis on the non-ROS software controllers. The performance of this evaluation shows that future work can be done on optimizing the path-finding algorithm. Some possible ideas may include but are not limited to parallelizing intracomponent analyses and other optimization techniques that reduces traversals of common subpaths.

The third area of improvement is to extend CFG-based checking to incorporate the function-call stack of called function. As discussed, the existing implementation of CFG-based checking considers all instances of a function call to be valid destinations for the function return, in order that the return node is connected to the correct calling context. The set of valid function return destinations could be narrowed by implementing a function-call stack, that records the sequence of function calls associated with a CFG path. The

stack contents could be used when associating a function return with its corresponding caller.

The fourth area of improvement is to extend Rex to support the extraction of ROS2 communication facts. As discussed previously, ROS2 uses a slightly different syntax in declaring publisher and subscriber nodes. By modifying Rex to consider these changes, we can apply our analyses to the additional systems.

References

[1] Hiralal Agrawal and Joseph R Horgan. "Dynamic Program Slicing". *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[2] Frances E Allen. "Control flow analysis". *ACM Sigplan Notices*, 5(7):1–19, 1970.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[4] Xiang Chen and Joanne M Atlee. Variability-Aware Neo4j for Analyzing a Graphical Model of a Software Product Line. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 307–318. IEEE, 2023.

[5] Paul Clements and Linda Northrop. *Software Product Lines*. Addison-Wesley Boston, 2002.

[6] Assylbek Dakibay. "Autonomous Driving: Baseline Autonomy". Master's thesis, University of Waterloo, 2017.

[7] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable Source Code Queries with Datalog. In *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*, pages 2–27. Springer, 2006.

[8] Nathan Hawes, Ben Barham, and Cristina Cifuentes. Frappé: Querying the Linux Kernel Dependency Graph. In *Proceedings of the GRADES'15*, pages 1–6. 2015.

[9] Alfred V Hoe, Ravi Sethi, and Jeffrey D Ullman. *Compilers—Principles, Techniques, and Tools*. Pearson Education, 2007.

[10] Sayali Kate, Michael Chinn, Hongjun Choi, Xiangyu Zhang, and Sebastian Elbaum. "PHYSFRAME: Type Checking Physical Frames of Reference for Robotic Systems". In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 45–56, 2021.

[11] Donglin Liang and Mary Jean Harrold. "Efficient Points-to Analysis for Whole-Program Analysis". *ACM SIGSOFT Software Engineering Notes*, 24(6):199–215, 1999.

[12] LLVM Projects. "Clang: A C Language Family Frontend for LLVM". https://clang.llvm.org/, 2024. Accessed: (March 2024).

[13] Amgad Madkour, Walid G Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. "A Survey of Shortest-Path Algorithms". *arXiv preprint arXiv:1705.02044*, 2017.

[14] Bryan J Muscedere, Robert Hackman, Davood Anbarnam, Joanne M Atlee, Ian J Davis, and Michael W Godfrey. "Detecting Feature-Interaction Symptoms in Automotive Software Using Lightweight Analysis". In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–185. IEEE, 2019.

[15] Bryan James Muscedere. "Detecting Feature-Interaction Hotspots in Automotive Software using Relational Algebra". Master's thesis, University of Waterloo, 2018.

[16] Neo4j, Inc. "APOC (Awesome Procedures On Cypher)". https://neo4j.com/labs/apoc/4.4/, 2021. Accessed: (March 2024).

[17] Neo4j, Inc. "Neo4j Graph Platform". https://neo4j.com/, 2021. Accessed: (March 2024).

[18] Neo4j, Inc. "Cypher Query Language". https://neo4j.com/developer/cypher/, 2023. Accessed: (March 2024).

[19] Open Robotics. "ROS - Robot Operating Systems". https://ros.org/, 2021. Accessed: (March 2024).

[20] Open Robotics. "Welcome to the ROS Community". https://www.ros.org/blog/community/, 2021. Accessed: (March 2024).

[21] Rudolf Ramler, Georg Buchgeher, Claus Klammer, Michael Pfeiffer, Christian Salomon, Hannes Thaller, and Lukas Linsbauer. Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases. In *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud: 11th International Conference, SWQD 2019, Vienna, Austria, January 15–18, 2019, Proceedings 11*, pages 125–148. Springer, 2019.

[22] Sandra Rapps and Elaine J. Weyuker. "Selecting Software Test Data Using Data Flow Information". *IEEE Transactions on Software Engineering*, (4):367–375, 1985.

[23] André Santos, Alcino Cunha, and Nuno Macedo. Static-Time Extraction and Analysis of the ROS Computation Graph. In *2019 Third IEEE international conference on robotic computing (IRC)*, pages 62–69. IEEE, 2019.

[24] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 196–206, 2016.

[25] Ramy Shahin, Murad Akhundov, and Marsha Chechik. Annotative Software Product Line Analysis Using Variability-Aware Datalog. *IEEE Transactions on Software Engineering*, 49(3):1323–1341, 2022.

[26] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for Fast and Easy Program Analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010.

[27] University of Waterloo. Autonomous Driving with Autonomoose. https://www.autonomoose.net, 2016. Accessed: (March 2024).

[28] University of Waterloo. WISE Automated Driving System and UW Moose. https://uwaterloo.ca/waterloo-intelligent-systems-engineering-lab/projects/wise-automated-driving-system-and-uw-moose, 2024. Accessed: (March 2024).

[29] Raoul-Gabriel Urma and Alan Mycroft. Source-code Queries with Graph Databases—with Application to Programming Language Usage and Evolution. *Science of Computer Programming*, 97:127–134, 2015.

[30] Nicolo Valigi. "Lessons Learned Building a Self-Driving Car on ROS". *ROSCon Madrid*, 2018.

[31] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005.

[32] Thomas Witte and Matthias Tichy. Checking Consistency of Robot Software Architectures in ROS. In *Proceedings of the 1st International Workshop on Robotics Software Engineering*, pages 1–8, 2018.

APPENDICES

Appendix A

Experiment Setup

A GitHub repository[1] is used to store all scripts and Cypher files used to perform the analyses discussed in Chapter 4. This repository is organized into four directories:

- `cypher-files/`: the Cypher queries for each of the analyses that performed in Chapter 4.
- `factbases/`: the factbases in CSV format for the four open-source projects used in the experiment
- `python-scripts/`: the python scripts to import CSV node and edge files into Neo4J databases
- `setup-scripts/`: bash scripts that configure the experiment environment and the JSON files that store Neo4J settings, input and output file directory information for running the Python scripts

In Chapter 4, cross-component analyses use a two-phased approach, and the remaining analyses are intra-component which can be completed with a single query. Table A.1 provides a summary of the different queries and the settings.

In general, a two-phased analysis is completed by running three Python scripts. The `phase1.py` obtains sub-query results for the component-based analyses. The `phase2.py` script is executed immediately after `phase1.py` and it outputs the system-wide results.

[1] https://github.com/xyfke/cfg-factbase-analyses

Table A.1: Queries Settings

Query Name	**Location**	**Type**	**Has Prefix/Suffix?**	**Multiple Paths?**
intercomponent-based communication	ros-cypher/inter-comm	two-phased	No	No
loop detection	ros-cypher/loop-detect	two-phased	No	No
multiple publishers	ros-cypher/multi-pub	single query	-	Yes
race conditions	ros-cypher/race-cond2	two-phased	No	Yes
behaviour alteration	ros-cypher/beh-alt	two-phased	Yes	No
publisher alteration	ros-cypher/pub-alt	two-phased	Yes	No
intracomponent call graph	non-ros-cypher/call-graph	single query	-	No
inter-component call graph	ros-cypher/call-graph	two-phased	Yes	No
recursion	non-ros-cypher/recursion	single query	-	No
variable traingle	non-ros-cypher/triangle	single query	-	No

Lastly, the `combinePath.py` combines the results from phase-1 and phase-2 analyses to produce the overall analysis path. On the other hand, a single-query analysis is completed by running the `query-script.py` file in the repository.

Appendix B

Other CFG Extensions

In Section 4.1, we discussed briefly how a fact is appended to a CFG-checked path. Some diagrams are not displayed due to similar resemblance to other diagrams and space constraints, hence the following section discusses the edge cases for each type of fact and the complete diagrams for appending these facts.

B.1 call

A *call* fact typically represents function calls, but we have added two variations of this fact to describe the relationships between functions that publish messages using ROS publishers, and ROS subscribers and their corresponding callback functions.

The first variation of a *call* (*f, p*) fact occurs when a function f publishes a message to a ROS topic with a ROS publisher p. The dashed portion in Figure B.1 shows the CFG checking required for appending this variation fact. It checks that there exist zero or more *nextCFGBlock* relations between the subpath's destination CFG block (cfg_i) to the publishing statement's source CFG block (cfg_n).

The second variation of a *call* (*s, f*) fact models the relationship between the ROS subscriber s and the callback function f which is automatically executed when this ROS subscriber receives a message from a ROS topic. There is no CFG-based checking when appending this variation because it represents a control flow entry to a new component, and there is no CFG edge from the publisher component to the subscriber component to check.

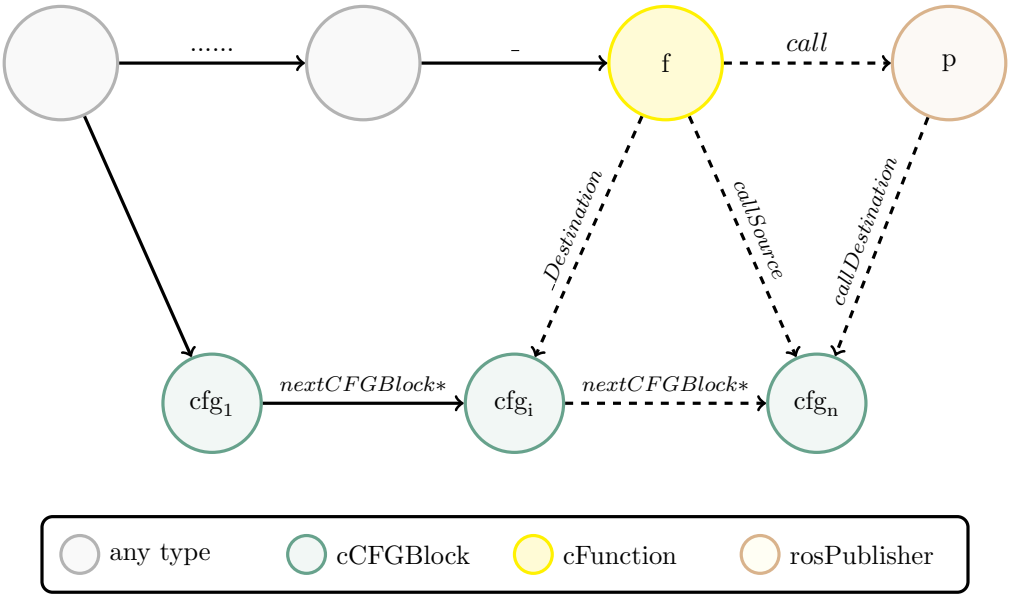


Figure B.1: Graphical pattern of a subquery involving the variation of a *call* fact, and its CFG-based check

B.2 write

A *write* (*f*, *v*) fact depicts a variable *v* has been modified within the body of function *f*. To append this fact to a CFG-checked path, it checks that there exist zero or more *nextCFGBlock* relations between the subpath's destination CFG block (cfg_i) to the variable assignment's source CFG block (cfg_n). The dashed portion of Figure B.2 depicts the relationships to be checked as a part of CFG-based checking.

B.3 publish

A *publish* (*p*, *t*) fact occurs when a ROS publisher *p* publishes to a ROS topic *t* in a `publish` statement. To append such a fact to a CFG-checked path, it checks that there exist zero or more *nextCFGBlock* relations between the subpath's destination CFG block (cfg_i) to the publishing statement's source CFG block (cfg_n). This check is depicted in the dashed portion of Figure B.3.

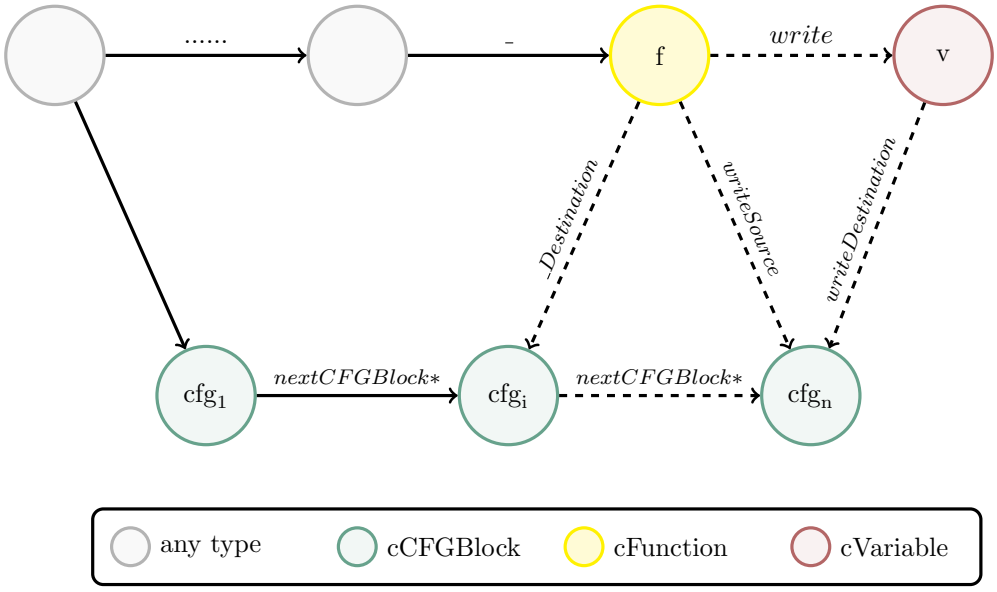


Figure B.2: Diagram for CFG extension of *write* fact

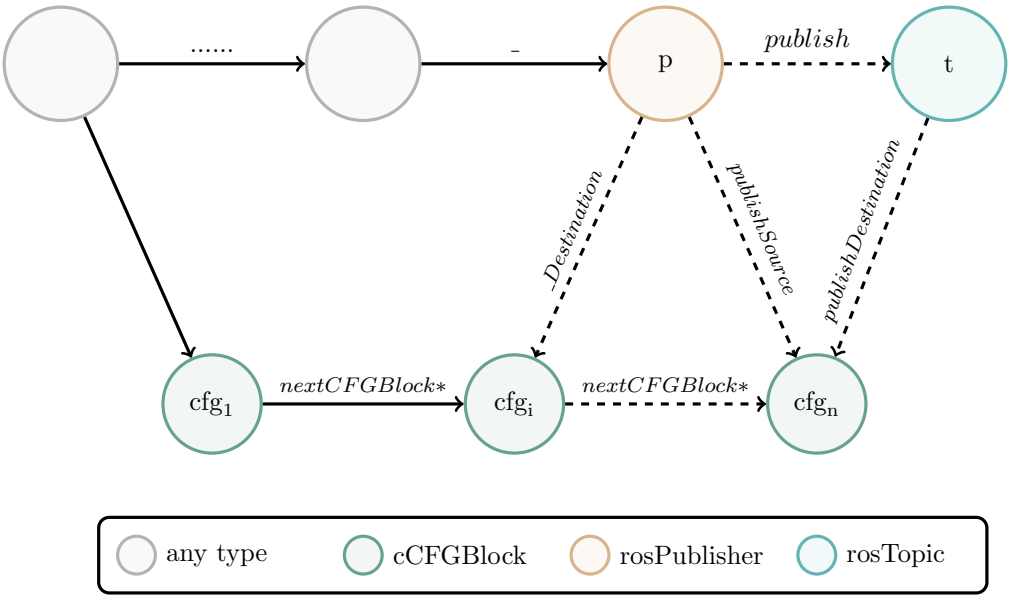


Figure B.3: Graphical pattern of a subquery involving a *publish* fact, and its CFG-based check

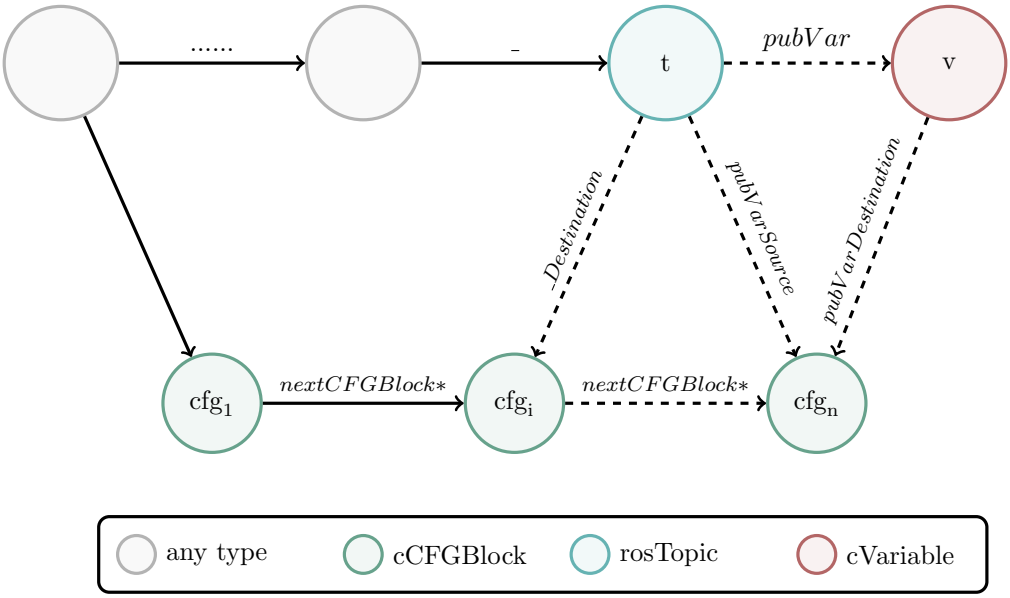


Figure B.4: Graphical pattern of a subquery involving a *pubVar* fact, and its CFG-based check

B.4 pubVar

A *pubVar* (v, t) fact occurs when a ROS message variable v is published to a ROS topic **t** in a `publish` statement. To append such a fact to a CFG-checked path, it checks that there exist zero or more *nextCFGBlock* relations between the subpath's destination CFG block (cfg_i) to the publishing statement's source CFG block (cfg_n). This check is depicted in the dashed portion of Figure B.4. As discussed previously, there is no intervening check to append this fact because the ROS message object contains multiple attributes that need to be modified before publishing this message.

B.5 varInfFunc

A *varInfFunc* fact represents a variable that determines whether or not a function is called in a control statement. The variation of this fact occurs when a function call with a return value r is in the decision of a control statement, whose body contains another function f. This variation is unique because there are exactly two *nextCFGBlock* relations between the destination CFG node of the checked subpath and the decision condition. The last edge

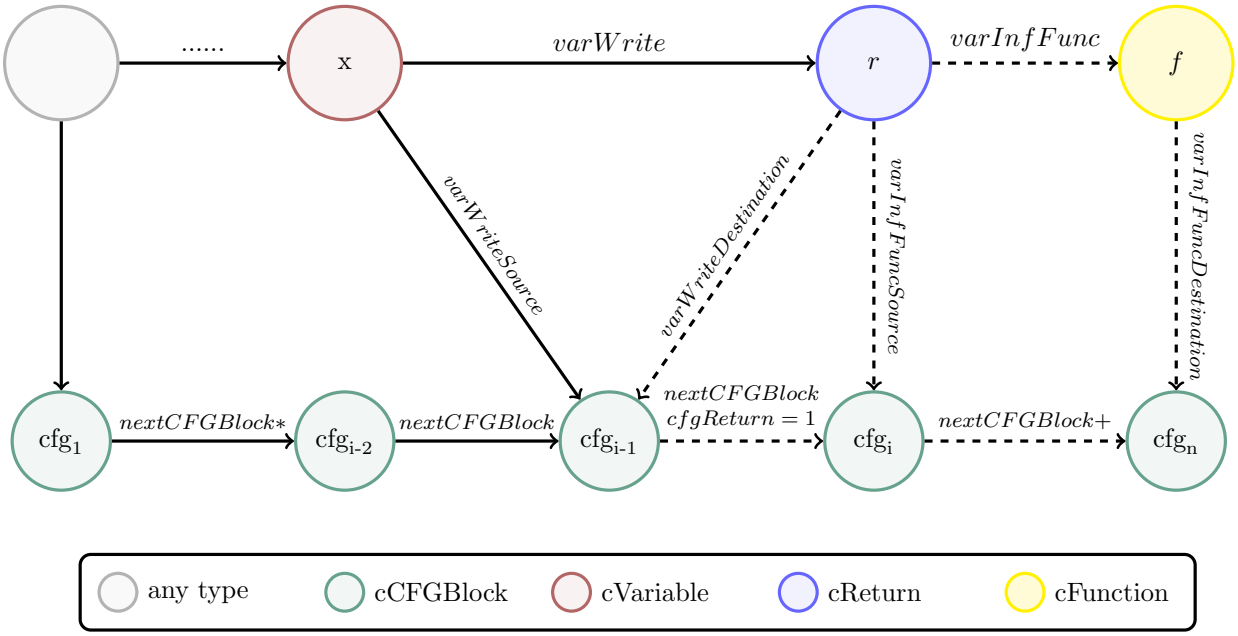


Figure B.5: Graphical pattern of a subquery involving the variation of a *varInfFunc* fact, and its CFG-based check

in the checked subpath must be a *varWrite* to a return node: (1) the CFG edge from the return statement's CFG block to the exit CFG block of the called function, and (2) the CFG edge from this exit block to the CFG block of the *varInfFunc*'s decision condition. The CFG check remains the same and Figure B.5 depicts this special case.

B.6 varInfluence

A *varInfluence* (*v,p*) fact occurs when the variable v in the decision condition of a control statement influences whether a ROS publish command p (rather than a function call) within the control structure is executed. The CFG check of appending such fact is similar to *varInfFunc* where it checks that there exists a sequence of one or more *nextCFGBlock* relations from the subpath's last destination CFG block (cfg_i) to the *varInfluence*'s source CFG block (cfg_{n-1}), with no intervening assignment to the decision variable (as shown in Figure B.6).

The one variation of a *varInfluence* (r, p) fact is also similar to the variation of *varInfFunc* fact. It occurs when a function call with a return value r is a part of the decision

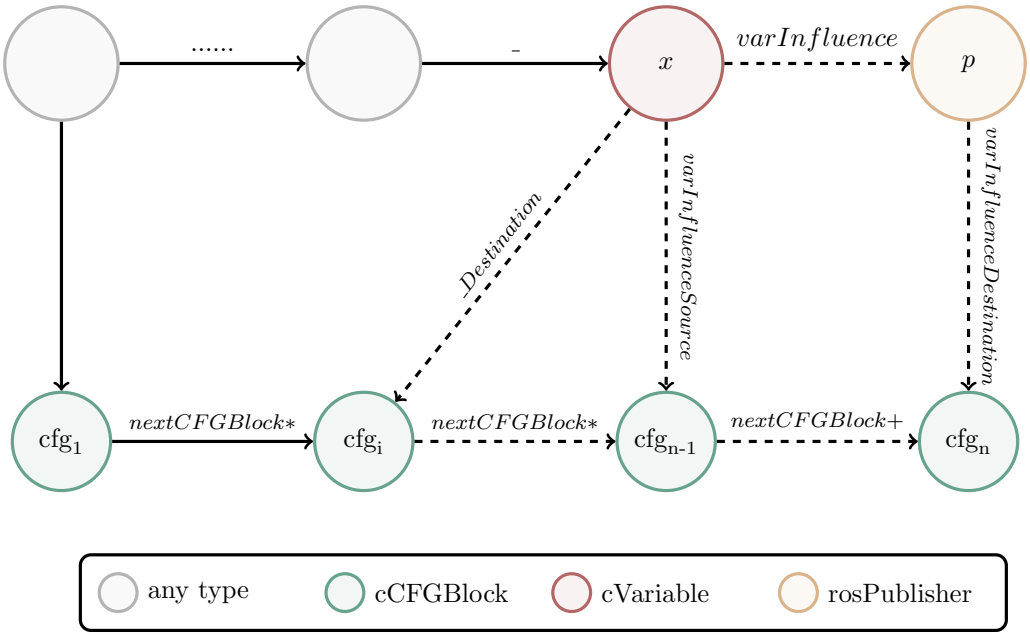


Figure B.6: Graphical pattern of a subquery involving a *varInfluence* fact, and its CFG-based check

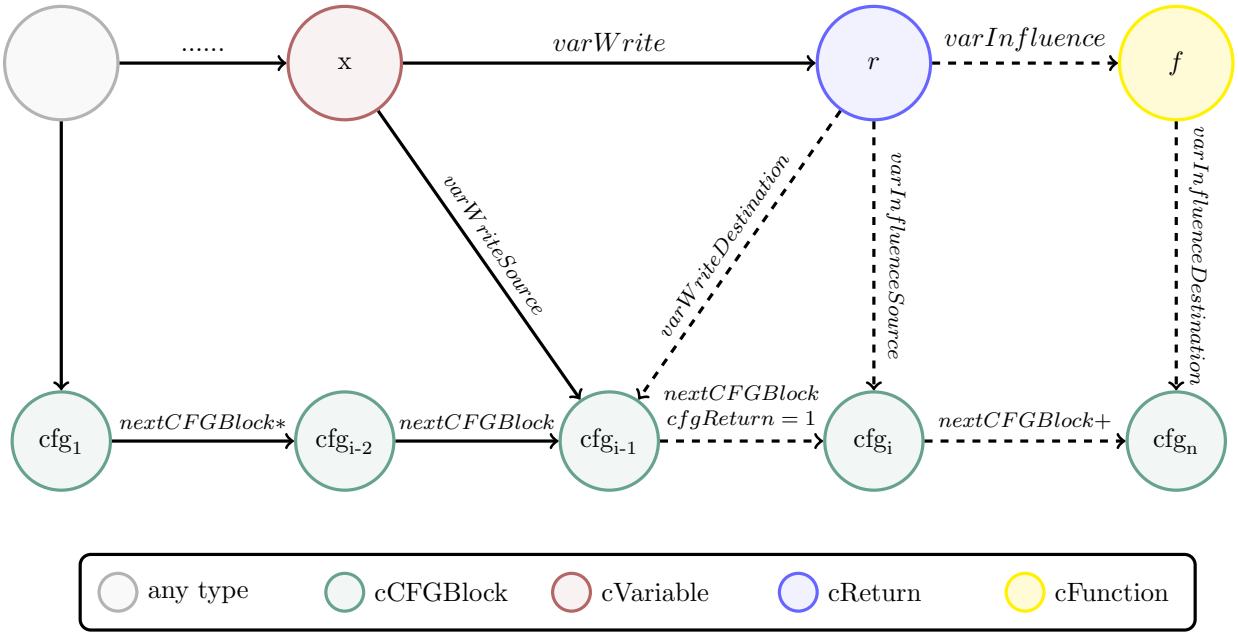


Figure B.7: Graphical pattern of a subquery involving the variation of a *varInfluence* fact, and its CFG-based check

condition. The CFG check for this variation consists of two *nextCFGBlock* relations between the destination CFG node of the checked subpath and the decision condition as shown in Figure B.7.

Appendix C

Neo4J Queries of Analyses

Chapter 5 briefly discusses all the analyses that we performed to test the performance and the amount of filtering achieved by comparing CFG-checked analyses against baseline analyses without CFG-based testing. The query statements and their respective graphical representations are presented in this Appendix section.

C.1 Inter-component Call Graph

Inter-component call-graph analysis is a control-flow query that reports the paths of function calls across multiple components. Each call-graph path contains at least one or more pairs of ROS publishers/subscribers, which represent communications between components. The graphical pattern of this query analysis is shown in Figure C.1.

Figure C.2 to C.5 show the Cypher queries for component-based and system-wide analyses. Figure C.2 shows the prefix subquery for an intracomponent call-graph analysis. The `MATCH` statements in Lines 1 and 2 retrieve all possible starting facts (any function nodes) and ending facts (i.e. any publishing edges) in the component factbase. For each pair of start and end facts, it uses the *apoc.cfgPath.rosFindPaths* function to return CFG-checked call paths within the factbase. In Figure C.2, *f1* is passed as a *start* node and *pub* is passed as an *endE* edge. The *relSeq* parameter in Line 4 specifies the accepted relations in the result paths, which is one or more *call* relationships; and the *filter* parameter in Line 5 serves as a node filter, indicating that only *function* and ROS *publisher* nodes are accepted when constructing the query results. The *config* describes the pattern between every *call* relation's CFG nodes: Lines 7-10 indicate that every *call* relationship from a function node

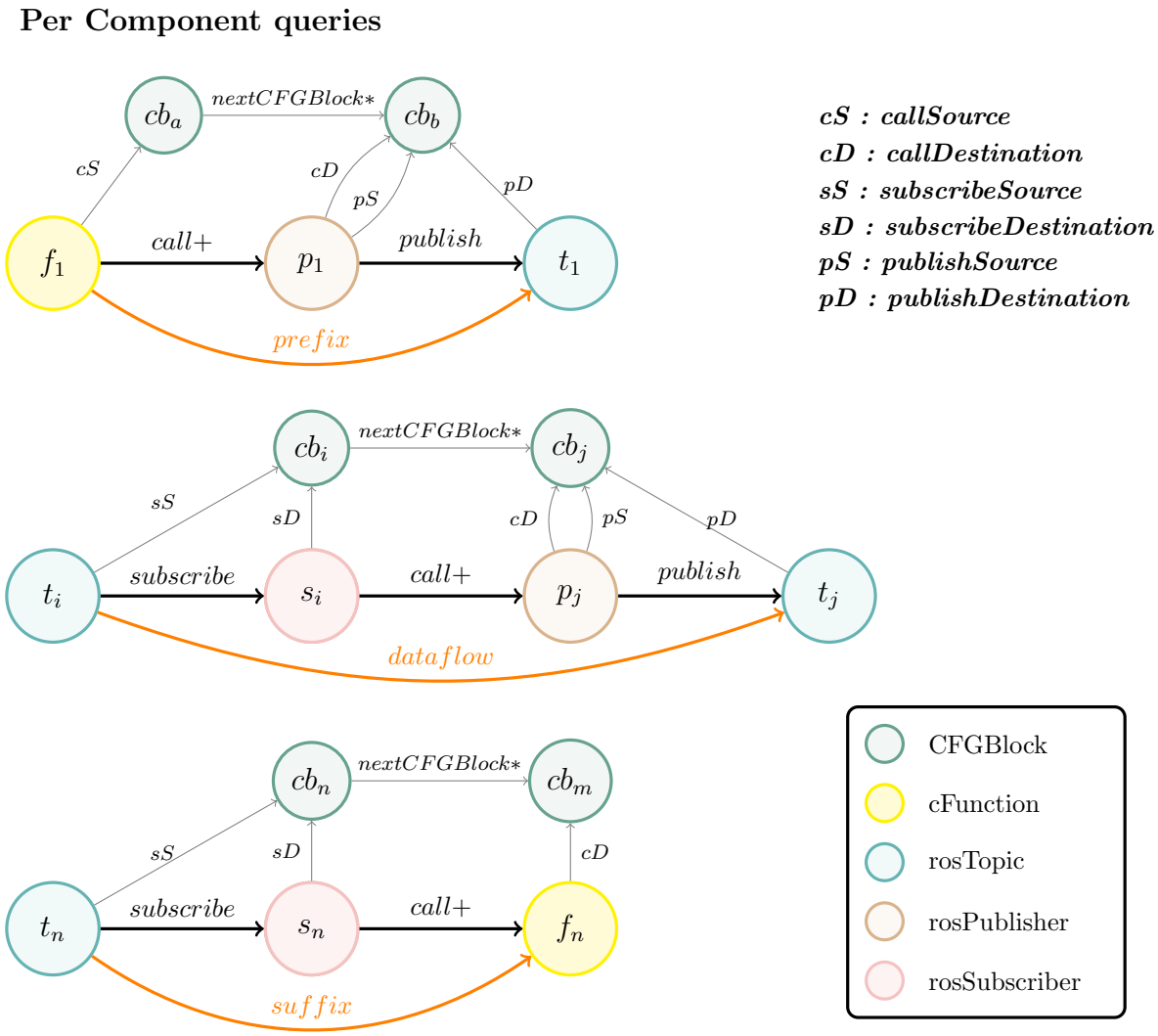


Figure C.1: The graphical pattern for an incremental cross-component call graph query. The top half of the figure depicts the intracomponent query patterns and the bottom half of the figure depicts the system-wide and inter-component query pattern (for ROS applications)

to another function node involves two CFG blocks, which are joined together by a single *nextCFGBlock* relation with an attribute of *cfgInvoke*. After running the APOC query, the results are returned as a list of paths, and Lines 12-13 retrieve each path from the list and print it out in the console of Neo4J.

```
1  // Retrieve all pairs of starting function node and ending publish edge of the
   ↪  prefix inter-component call graph query
2  MATCH (f1:cFunction)
3  MATCH (:rosPublisher)-[pub:publish]->(:rosTopic)
4  // Find all shortest paths consisting of one or more call edges for each pair of
   ↪  start node and end edge
5  WITH *, apoc.cfgPath.rosFindPaths(f1, {
6      relSeq : "call+",
7      filter : "cFunction,rosPublisher",
8      endE : pub,
9      config : [
10         {name : "call", startLabel : "cFunction", endLabel : "cFunction",
11         attribute : "cfgInvoke", length : "1"}
12     ]
13 }) As paths
14 // Return and display each path
15 UNWIND paths As path
16 RETURN DISTINCT path;
```

Figure C.2: Prefix query for ROS component call-graph.

Figures C.3 and C.4 present the intermediate call-graph subquery and the suffix call-graph subquery, respectively. The parameters used in the APOC functions are essentially the same, except that both queries start with a *subscribe* edge, and the suffix call-graph subquery result terminates with a function *call* whose destination function has no internal function call.

Lastly, Figure C.5 shows the system-wide call-graph query performed on the augmented system-wide factbase which includes summary facts produced by the intracomponent call-graph subqueries. The resulting paths are sorted based on the number of components involved in each path, with results involving the highest number of components reported first.

```
// Retrieve all pairs of starting subscribe edge and ending publish edge of the
↪  intermediate inter-component call graph query
MATCH (:rosTopic)-[sub:subscribe]->(:rosSubscriber)
MATCH (:rosPublisher)-[pub:publish]->(:rosTopic)
// Find all shortest paths consisting of one or more call edges for each pair of
↪  start and end edges
WITH *, apoc.cfgPath.rosFindPaths(sub, {
    relSeq : "call+",
    filter : "cFunction,rosPublisher",
    endE : pub,
    config : [
        {name : "call", startLabel : "cFunction", endLabel : "cFunction",
        attribute : "cfgInvoke", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.3: Intermediate query for ROS component call graph.

```
// Retrieve each starting subscriber call edge of the intermediate
↪  inter-component call graph query
MATCH (:rosTopic)-[sub:subscribe]->(:rosSubscriber)
// Find all shortest paths consisting of one or more call edges starting with a
↪  subscribe edge
WITH *, apoc.cfgPath.rosFindPaths(sub, {
    relSeq : "call+",
    filter : "cFunction",
    config : [
        {name : "call", startLabel : "cFunction", endLabel : "cFunction",
        attribute : "cfgInvoke", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.4: Suffix query for ROS component call graph.

```
// Find summary edge paths in descending order of number of components
MATCH path= (startPubTopic:rosTopic)-[:callflow*0..]->(endSubTopic:rosTopic)
RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.5: System-wide query for ROS cross-component call graph.

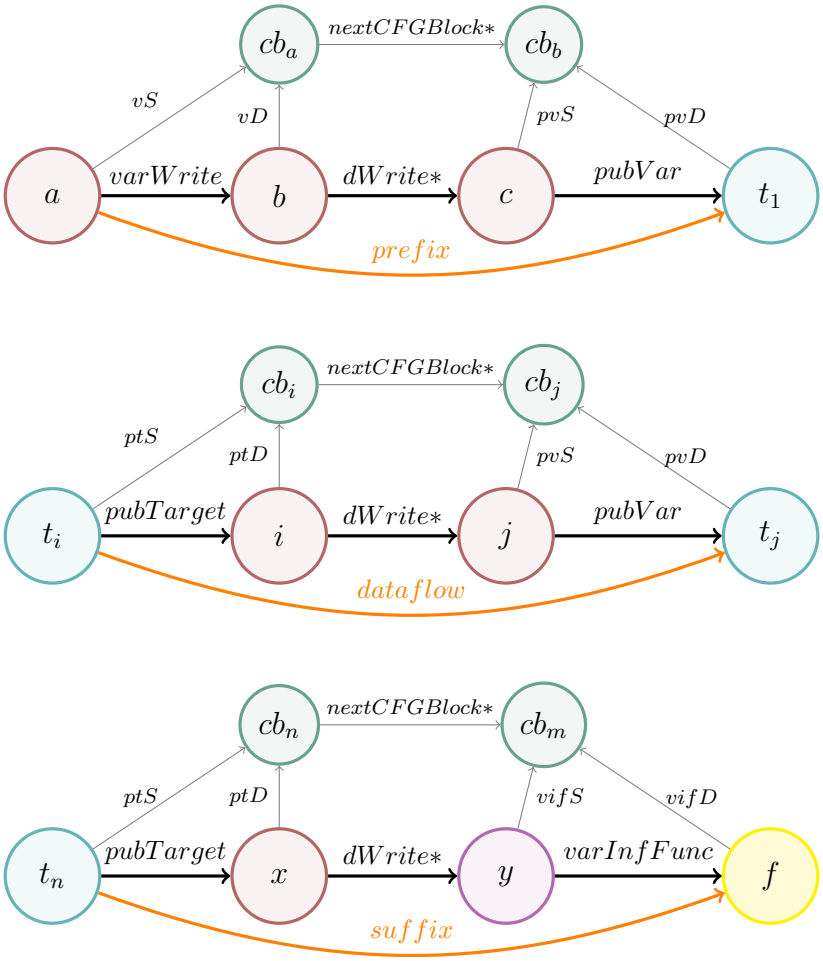
C.2 Behaviour Alteration Analyses

Behaviour alteration analysis is a feature-interaction query that detects unexpected behaviours in software applications. A behaviour alteration is initiated by a variable assignment that leads to modified behaviour in another component. Specifically, this query tracks how (1) some initial variable assignment impacts the values of other variables through a series of subsequent variable assignments, parameter passings, and function return assignments before publishing an impacted variable value to a ROS topic. (2) A different component that subscribes to this ROS topic receives the impacted variable value as an input message. The query then follows all dataflows (i.e. variable assignments, parameter passes, and function return assignments) from the input message variable to the point where the component publishes to another component where some variable whose value is impacted by the input message variable. (3) The dataflow progresses through one or more intermediate components in this manner until it reaches a control statement in some component where an impacted variable appears in the decision condition of the control statement and the body of the control statement includes a function call. Behaviour alteration paths that involve only two components are excluded from the analysis results because direct communication between two components is likely an intended behaviour of the application and therefore is not worthy of further examination.

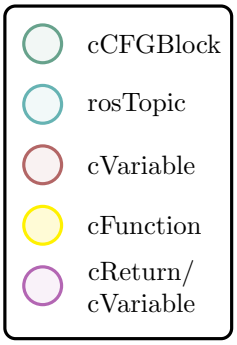
Figure C.6 shows the patterns among program elements used to find the behaviour-alteration paths with CFG-based checking. The upper half of Figure C.6 depicts the pattern of the prefix, intermediate, and suffix intracomponent subqueries that are performed on all component factbases. The bottom half of the figure depicts system-wide query performed on the full factbase augmented with summary facts representing intracomponent query results. Note that no CFG-based checking is performed on the system-wide query because it has already been done when performing the component-based subqueries. The Neo4J/APOC queries are provided in Appendix C.

Figures C.7 to C.10 show the Cypher queries for component-based and system-wide analyses. Figure C.7 shows the prefix subquery for a behaviour alteration analysis. The `MATCH` statements in Lines 1 and 2 retrieve all possible starting (any variable assignments) and ending points (i.e. any publishing message) in the component factbase. For each pair of start and end points, it uses the *apoc.cfgPath.rosFindPaths* function to return CFG-checked call paths within the factbase. Specifically, *vw* is passed as a *start* edge and *pv* is passed as an *endE* edge. The *relSeq* parameter in Line 4 specifies the accepted relations in the result paths, which is one or more dataflow relationships (i.e. a series of variable assignments, parameter passings, and function return assignments); and the *filter* parameter in Line 5 serves as a node filter, indicating that only variable and function return

Patterns for intracomponent queries



vS : varWriteSource
vD : varWriteDestination
pvS : pubVarSource
pvD : pubVarDestination
ptS : pubTargetSource
ptD : pubTargetDestination
vifS : varInfFuncSource
vifD : varInfFuncDestination
dWrite : varWrite/parWrite /retWrite



Pattern for system-wide queries



Figure C.6: Patterns for behaviour alteration queries for ROS applications

```
// Retrieve all pairs of starting variable assignments and ending message
↪  publish edge of the prefix behaviour alteration query
MATCH (:cVariable)-[vw:varWrite]->(:cVariable)
MATCH (:cVariable)-[pv:pubVar]->(:rosTopic)
// Find all shortest paths consisting of one or more dataflow edges for each
↪  pair of start and end edges
WITH *, apoc.cfgPath.rosFindPaths(vw, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : pv,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.7: Prefix query for ROS Behaviour Alteration.

nodes are accepted when constructing the sub-paths. Line 7 indicates that it will return all shortest paths described in Section 4.2.2 for every unique pair of start and endpoints. The *config* describes the pattern between the CFG nodes and the relation:

1. Lines 9-10 states that every *parWrite* relationship from a function return node to a variable node (i.e. that is a parameter) involves three CFG blocks joined together by two *nextCFGBlock* relationship, with the second one annotated with the attribute *cfgInvoke* (Specifically, it returns to the calling context, before entering the second called context).

2. Lines 11-12 indicate that every *parWrite* relationship from a variable to another variable node involves two CFG blocks which are joined together by a single relation, with an attribute of *cfgInvoke*.

3. Lines 13-14 describes the pattern for *retWrite* from a return node to a variable. It is similar to *parWrite* except that the *nextCFGBlock* relation between the two CFG nodes is annotated with *cfgReturn*.

4. Lines 15-16 describes the variation of *retWrite* from a function return node to another function return node. The CFG nodes and relations are the same as the case from Lines 13-14.

Even though the *varWrite* patterns are not described in the config section, it uses the default assumption that the relation is only associated with one CFG node (in that the source and destination CFG nodes are the same). After running the APOC query, the results are returned as a list of paths, and Lines 19-20 retrieve each path from the list and print it out in the console of Neo4J.

Figures C.8 and C.9 represent the intermediate behaviour alteration subquery and the suffix behaviour alteration subquery respectively. The parameters used in the APOC functions are essentially the same, except that both queries start with a *pubTarget* edge, and the suffix behaviour alteration subquery result terminates with a *varInfFunc* relationship (i.e. a variable is used in the decision condition of a control statement, whose body contains a function call).

Lastly, Figure C.10 shows the system-wide call-graph query performed on the augmented system-wide factbase that includes summary facts produced by the intracomponent call-graph subqueries. The resulting paths are sorted in descending order concerning the number of components affected in each path.

```
// Retrieve all pairs of starting message receiving and ending message publish
↪  edge of the intermediate behaviour alteration query
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH (:cVariable)-[pv:pubVar]->(:rosTopic)
// Find all shortest paths consisting of one or more dataflow edges for each
↪  pair of start and end edges
WITH *, apoc.path.cfgValidatedPath(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : pv,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.8: Intermediate query for ROS Behaviour Alteration.

```
// Retrieve all pairs of starting message receiving and ending varInfFunc edge
    of the suffix behaviour alteration query
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH ()-[vif:varInfFunc]->(:cFunction)
// Find all shortest paths consisting of one or more dataflow edges for each
    pair of start and end edges
WITH *, apoc.cfgPath.rosFindPaths(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : vif,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"},
        {name : "varInfFunc", startLabel : "cVariable", endLabel : "cFunction",
        length : "+"},
        {name : "varInfFunc", startLabel : "cReturn", endLabel : "cFunction",
        length : "+"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.9: Suffix query for ROS Behaviour Alteration.

```
// Find summary edge paths in descending order of number of components
MATCH path= (startPubTopic:rosTopic)-[:dataflow*1..]->(endSubTopic:rosTopic)
RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.10: Full system query for ROS Behaviour Alteration.

C.3 Publisher Alteration Analyses

A **publisher alteration analysis** is a variation of a behaviour alteration analysis. This type of query reports instances where a variable assignment in one component indirectly causes another component to publish a message to a topic. The prefix, intermediate, and full system publisher alteration queries are essentially the same as behaviour alteration as shown in Figures C.11, C.12, and C.14. The only difference lies in the suffix subquery shown in Figure C.13, where the endpoint is a *varInfluence* instead of a *varInfFunc* relationship. The CFG nodes and edges for *varInfluence* are the same as *varInfFunc*, as described in Lines 17-20. The graphical representation of this query is shown in Figure C.15.

```
1  // Retrieve all pairs of starting variable assignments and ending message
   ↪  publish edge of the prefix publisher alteration query
2  MATCH (:cVariable)-[vw:varWrite]->(:cVariable)
3  MATCH (:cVariable)-[pv:pubVar]->(:rosTopic)
4  // Find all shortest paths consisting of one or more dataflow edges for each
   ↪  pair of start and end edges
5  WITH *, apoc.cfgPath.rosFindPaths(vw, {
6      relSeq : "varWrite|parWrite|retWrite*",
7      filter : "cVariable,cReturn",
8      endE : pv,
9      shortest : true,
10     config : [
11         {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
12         attribute : "cfgReturn,cfgInvoke", length : "2"},
13         {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
14         attribute : "cfgInvoke", length : "1"},
15         {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
16         attribute : "cfgReturn", length : "1"},
17         {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
18         attribute : "cfgReturn", length : "1"}
19     ]
20 }) As paths
21 // Return and display each path
22 UNWIND paths As path
23 RETURN DISTINCT path;
```

Figure C.11: Prefix query for ROS Publisher Alteration.

```
// Retrieve all pairs of starting message receiving and ending message publish
↪  edge of the intermediate publisher alteration query
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH (:cVariable)-[pv:pubVar]->(:rosTopic)
// Find all shortest paths consisting of one or more dataflow edges for each
↪  pair of start and end edges
WITH *, apoc.path.cfgValidatedPath(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : pv,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.12: Intermediate query for ROS Publisher Alteration.

```
// Retrieve all pairs of starting message receiving and ending varInfluence edge
↪  of the suffix publisher alteration query
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH ()-[vi:varInfluence]->(:rosPublisher)
WITH *, apoc.cfgPath.rosFindPaths(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn,rosPublisher",
    endE : vi,
    shortest : true,
    // Find all shortest paths consisting of one or more dataflow edges for each
↪  pair of start and end edges
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"},
        {name : "varInfluence", startLabel : "cVariable", endLabel :
↪  "rosPublisher",
        length : "+"},
        {name : "varInfluence", startLabel : "cReturn", endLabel :
↪  "rosPublisher",
        length : "+"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.13: Suffix query for ROS Publisher Alteration.

```
1 // Find summary edge paths in descending order of number of components
2 MATCH path= (startPubTopic:rosTopic)-[:dataflow*1..]->(endSubTopic:rosTopic)
3 RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.14: Full system query for ROS Publisher Alteration.

C.4 Inter-Component-Based Communication

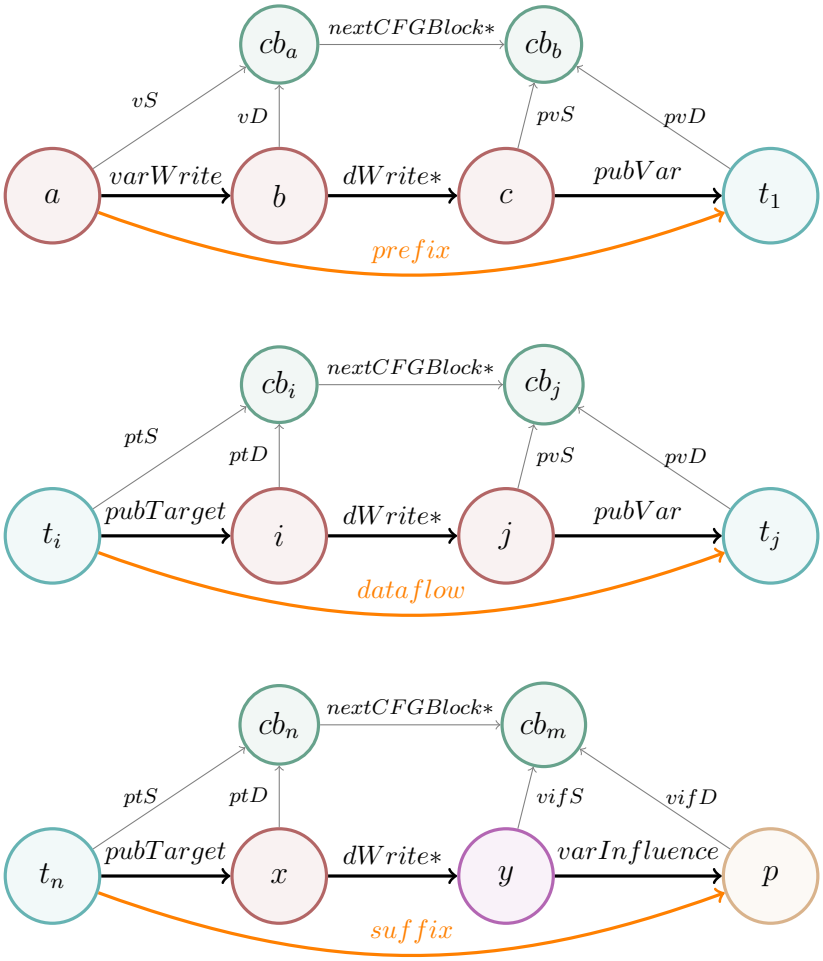
Inter-component-based communication analysis detects communications between two or more components through intracomponent and inter-component dataflows. The Cypher queries for this analyses slightly differs from the behaviour alteration and publisher alteration analyses described above, where it does not perform prefix and suffix subqueries in the phase-1 component-based analyses. Instead, it is done in the system-wide analyses in phase 2. In addition, this analysis does not require the combining of paths after phase 2. Figures C.16 shows the intermediate dataflow subquery which is essentially the same as the one for behaviour alterations and publisher alterations.

The system-wide analysis query for phase 2 is shown in Figure C.17. Line 1 represents the start point which is a *pubVar* relation and Line 2 represents the endpoint which is a *pubTarget* relation. Lines 3 to 7 look for the summary relations created from the results of phase-1 analysis and ensure that each analysis result path has no repeating components. Lastly, Lines 8-9 sort the results and return them by the total number of components invovled. A graphical representation of this query is shown in Figure C.18.

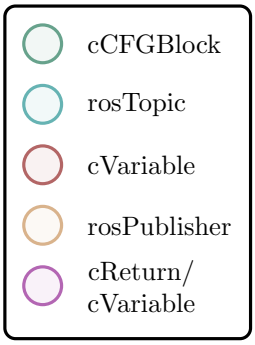
C.5 Loop Detection

Loop detection analysis reports instances of direct or indirect self-communications. This type of analysis poses the exact same queries as the intercomponent communication analysis. The only difference is that it reports only those results where the first and last components are equal. The phase-1 subquery for this analysis is the same as inter-component communication, as shown in Figure C.19. The system-wide query for phase-2, shown in Figure C.20, is slightly different where it checks that the starting and ending components are the same. This is shown in Line 2 in the figure.

Patterns for intracomponent queries



vS : varWriteSource
vD : varWriteDestination
pvS : pubVarSource
pvD : pubVarDestination
ptS : pubTargetSource
ptD : pubTargetDestination
vfS : varInfluenceSource
vfD : varInfluenceDestination
dWrite : varWrite/parWrite /retWrite



Pattern for system-wide query



Figure C.15: Patterns for publisher alteration queries for ROS applications

```
// Retrieve all pairs of starting message receiving and ending message publish
↪  edge of the intermediate inter-component-based query
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH (:cVariable)-[pv:pubVar]->(:rosTopic)
// Find all shortest paths consisting of one or more dataflow edges for each
↪  pair of start and end edges
WITH *, apoc.path.cfgValidatedPath(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : pv,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.16: Intermediate query for ROS Inter-Component-Based communication.

```
// Find send and receive components
MATCH pubVarPath = (a)-[:pubVar]->(startPubTopic:rosTopic)
MATCH pubTargetPath = (endSubTopic:rosTopic)-[:pubTarget]->(b:cVariable)
// Look for inter-component-based communication paths with no repeating components
MATCH df=(startPubTopic)-[:dataflow*0..]->(endSubTopic)
UNWIND relationships(df) As r
WITH pubVarPath, pubTargetPath, a, b, df, COLLECT(DISTINCT r.compName) As compNames
WHERE size(compNames) = length(df) AND (NOT a.compName IN compNames)
    AND (NOT b.compName IN compNames)
RETURN DISTINCT apoc.path.combine(apoc.path.combine(pubVarPath, df), pubTargetPath) As path
ORDER BY length(path) DESC;
```

Figure C.17: Full system query for ROS Inter-Component-Based Communication.

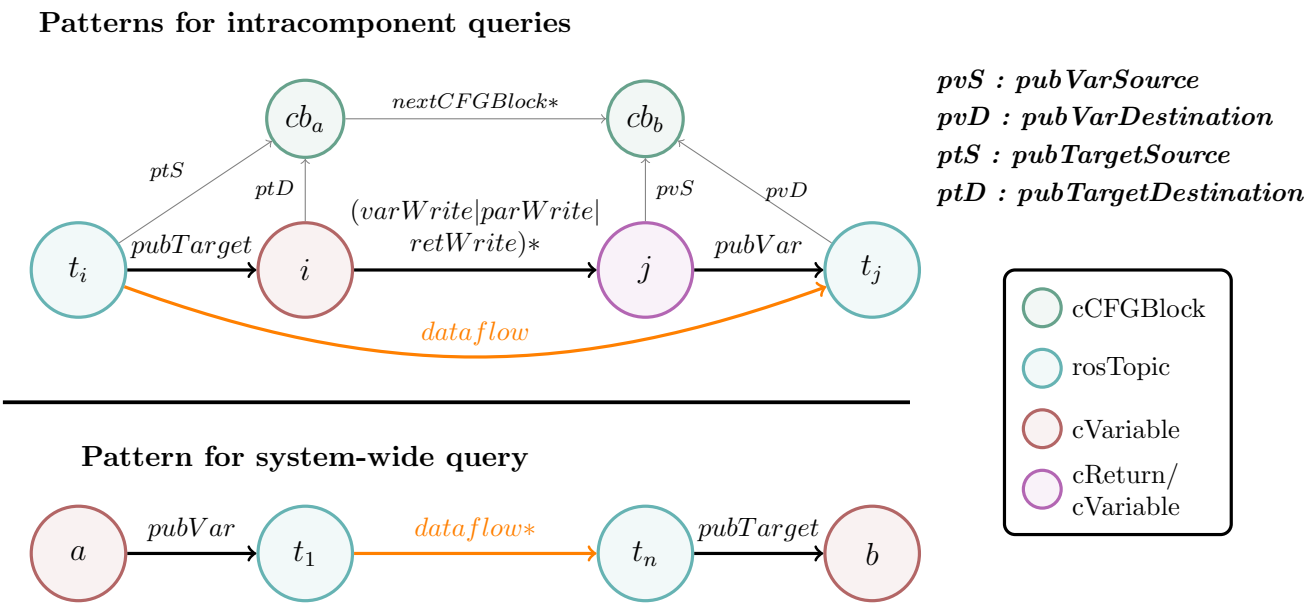


Figure C.18: Patterns for intercomponent communication queries for ROS applications

```
// Retrieve all pairs of starting message receiving and ending message publish
↪  edge of the intermediate loop detection query
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH (:cVariable)-[pv:pubVar]->(:rosTopic)
// Find all shortest paths consisting of one or more dataflow edges for each
↪  pair of start and end edges
WITH *, apoc.path.cfgValidatedPath(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : pv,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
// Return and display each path
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.19: Intermediate query for ROS Loop Detection.

```
// Find send and receive components and check that they are the same
MATCH path = (a:cVariable)-[:pubVar]->(:rosTopic)-[:dataflow*0..]->\\
    (:rosTopic)-[:pubTarget]->(b:cVariable)
WHERE a.compName = b.compName
RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.20: Full system query for ROS Loop Detection.

C.6 Multiple Publishers

Multiple publishers analysis detects when multiple publishers pass messages that lead to the same callback function in a ROS subscriber. This analysis has no CFG check because it only works with cross-component facts, hence it can be performed with a single query reporting all the results, shown in Figure C.21. Both Lines 1 and 2 finds publishing and subscribe components, with a *pubVar* followed by a *pubTarget* relationship. Figure C.22 shows the graphical representation of this query.

```
1 // Find different message variables received by the same callback parameter
2 MATCH path1=(a)-[:pubVar]->(t:rosTopic)-[:pubTarget]->(c)
3 MATCH path2=(b)-[:pubVar]->(t)-[:pubTarget]->(c)
4 WHERE a <> b AND a.compName <> b.compName
5 RETURN [path1, path2] As path;
```

Figure C.21: Full system query for ROS Multiple Publishers.

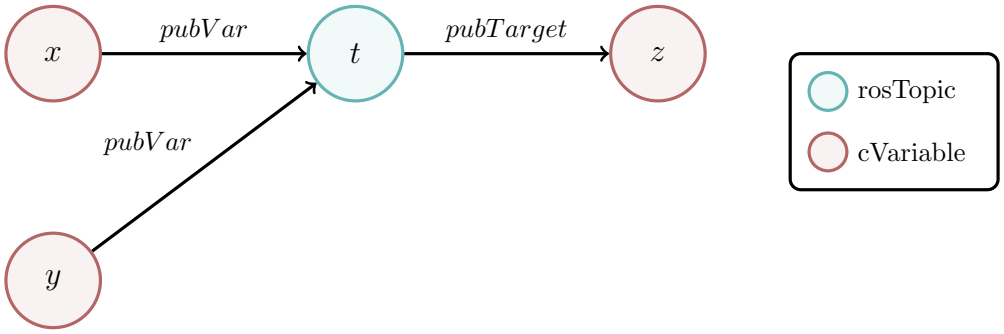


Figure C.22: Multiple publishers query

C.7 Race Conditions

An analysis similar to multiple publishers is the **race condition analysis**. This analysis detects when different publishers attempt to change the same variable in a subscriber component. This analysis detects when different publishers attempt to change the same variable in a subscriber component. This may cause nondeterministic behaviour if different orders of input messages lead to different system behaviours. Another potential problem

is that the processing of later messages can overwrite the processing of previously received messages. For this analysis, we use a two-phased approach without the prefix and suffix subqueries. The intermediate dataflow subquery is shown in Figure C.23, and it ends with a variable node.

```
// Track all messages that are written directly or indirectly to a variable
MATCH (:rosTopic)-[pt:pubTarget]->(:cVariable)
MATCH (b:cVariable)
WITH *, apoc.cfgPath.rosFindPaths(pt, {
    relSeq : "varWrite|parWrite|retWrite*",
    endN : b,
    shortest : true,
    filter : "cVariable,cReturn",
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.23: Intermediate query for ROS Race Conditions.

The system-wide query shown in Figure C.24 checks that message variables a and b are not the same, and there are in fact two topics trying to modified the same attribute or variable in component c. Figure C.25 shows the graphical representation of a race condition query.

```
// Find topics and sender that modify the same variable
MATCH path1 = (a)-[:pubVar]->(t1:rosTopic)-[:dataflow]->(c:cVariable)
MATCH path2 = (b)-[:pubVar]->(t2:rosTopic)-[:dataflow]->(c)
WHERE a <> b AND t1 <> t2
RETURN [path1, path2] As path;
```

Figure C.24: Full system query for ROS Race Conditions.

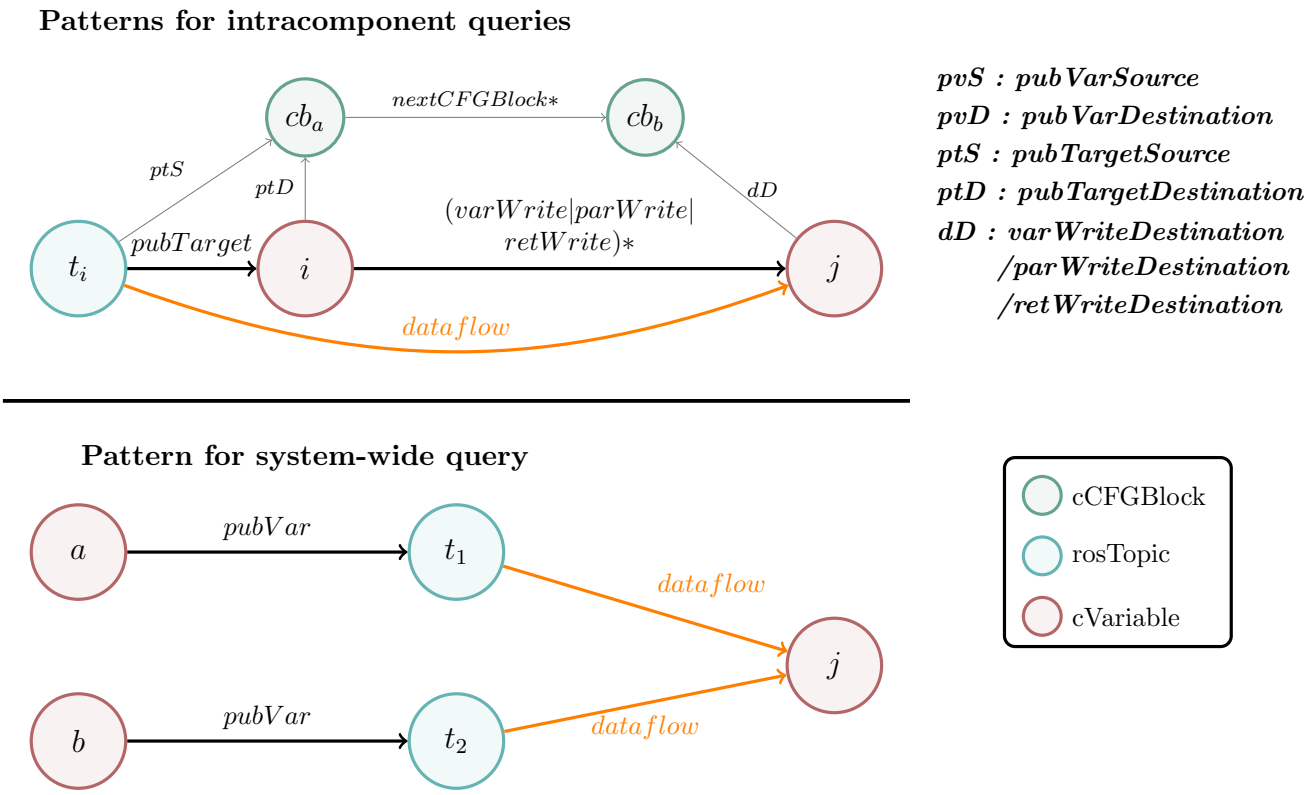


Figure C.25: Patterns for race condition queries for ROS applications

C.8 Intracomponent Call Graph

An **intracomponent call graph analysis** reports on paths of function calls that happen within the same component. This analysis is completed with a single query on the full system factbase, depicted in Figure C.26. Line 1 indicates the starting and passes into the APOC function and the config lines specify the CFG nodes and relationships that are associated with the *call* fact in the analysis results. Figure C.27 depicts the graphical representation of such an analysis.

```
1  // Find all intracomponent call graph with each starting function
2  MATCH (a:cFunction)
3  WITH *, apoc.cfgPath.rosFindPaths(a, {
4      relSeq : "call+",
5      filter : "cFunction",
6      config : [{
7          name : "call",
8          startLabel : "cFunction",
9          endLabel : "cFunction",
10         attribute : "cfgInvoke",
11         length : "1"
12     }]
13 }) As paths
14 // return results
15 UNWIND paths As path
16 RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.26: Query for Intracomponent Call Graph.

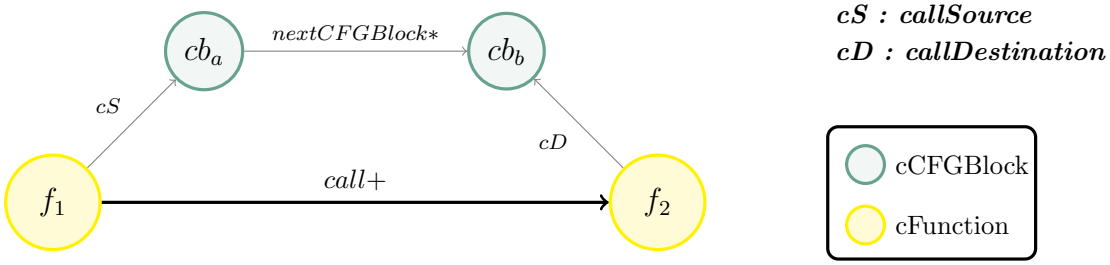


Figure C.27: Pattern for intracomponent call graph query for ROS applications

C.9 Recursion

A **recursion analysis** is an intracomponent analysis that reports on paths of function calls that directly or indirectly call themselves in a program. Figure C.28 depicts the query of this analysis. It is almost the same as intracomponent call graph analysis, except we check that the start and endpoints are the same function node. The graphical representation for a CFG-checked recursion analysis is shown in Figure C.29.

```
// Find each function that directly and indirectly call itself
MATCH (a:cFunction)
WITH *, apoc.cfgPath.rosFindPaths(a, {
    relSeq : "call+",
    endN : a,
    config : [{
        name : "call",
        startLabel : "cFunction",
        endLabel : "cFunction",
        attribute : "cfgInvoke",
        length : "1"
    }]
}) As paths
UNWIND paths As path
RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.28: Query for Recursion.

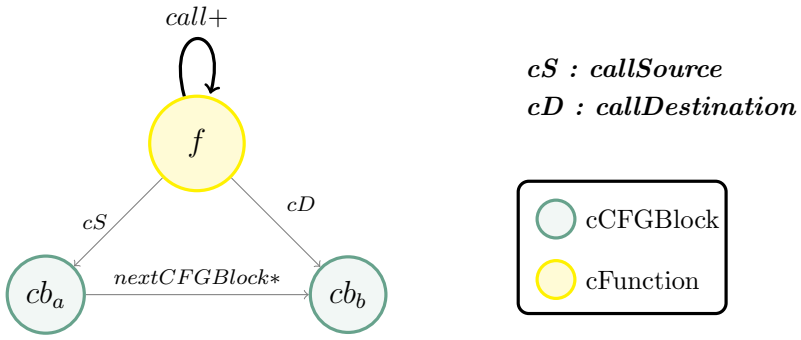


Figure C.29: Pattern for recursion query for ROS applications

C.10 Variable Triangle Communication

A **variable triangle communication analysis** is an intracomponent analysis that reports on variables that write to themselves after being used in a variable or parameter assignment to another variable. The Cypher query for this analysis is depicted in Figure C.30. Specifically, it finds a path with a length of three relations where the variable writes to itself. Lines 2 and 3 indicate that the start and end nodes are the same. Figure C.31 shows the graphical representation of this query.

```
1  // find instances of variables that write to itself
2  MATCH (a:cVariable)
3  WITH *, apoc.cfgPath.rosFindPaths(a, {
4      endN : a,
5      relSeq : "varWrite|parWrite,varWrite|parWrite,varWrite|parWrite",
6      config : [
7          {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
8          attribute : "cfgReturn,cfgInvoke", length : "2"},
9          {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
10         attribute : "cfgInvoke", length : "1"},
11         {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
12         attribute : "cfgReturn", length : "1"},
13         {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
14         attribute : "cfgReturn", length : "1"}
15     ]
16 }) As paths
17 UNWIND paths As path
18 RETURN DISTINCT path;
```

Figure C.30: Query for Variable Triangle Communication.

C.11 Behaviour Alteration Analyses for Non-ROS Applications

The Cypher/APOC query of non-ROS behaviour-alteration analyses is quite similar to the ROS analyses because everything remains the same except for the entry and exit points of

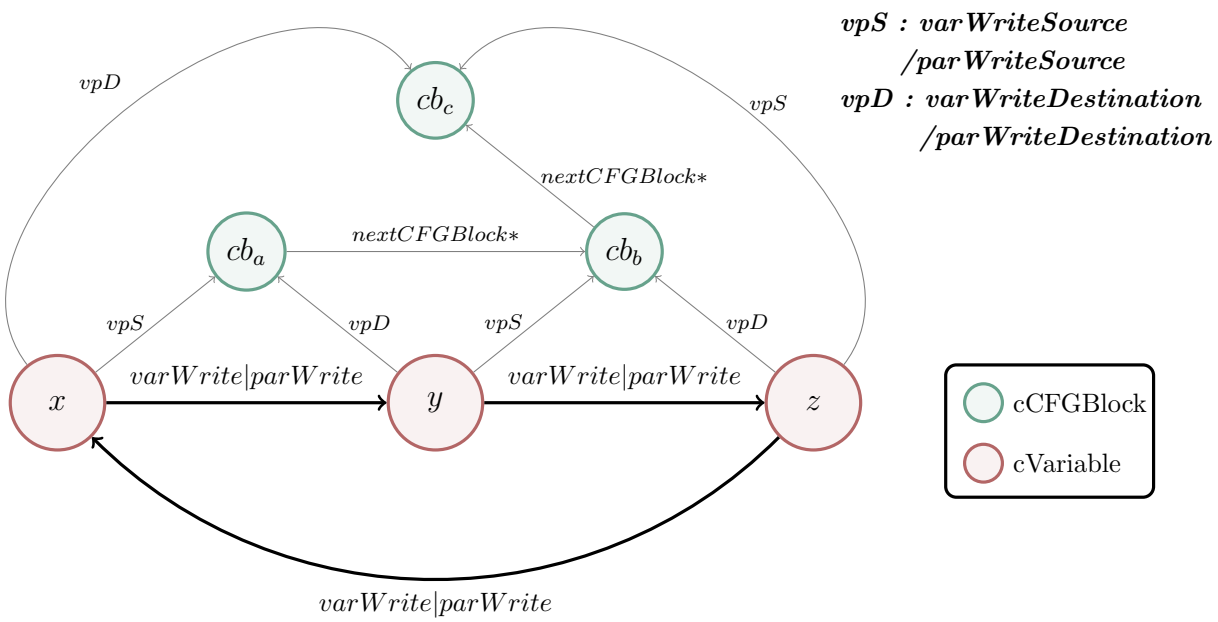


Figure C.31: Pattern for triangle variable communication query for ROS applications

each component-based subqueries. The component-based subqueries are shown in Figures C.32 to C.34. The starting fact is either a *varWriteIn* or *parWriteIn* or *retWriteIn* relationship. The ending fact for each of these queries is always a *varWriteOut* or *parWriteOut* or *retWriteOut* relationship. The system-wide query is shown in Figure C.35.

```
MATCH (:cVariable)-[vw:varWrite]->(:cVariable)
MATCH ()-[out:parWriteOut|retWriteOut]->()
WITH *, apoc.cfgPath.nonRosFindPaths(vw, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,return",
    endE : out,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.32: Prefix query for Non-ROS Behaviour Alteration.

```
MATCH ()-[inE:parWriteIn|retWriteIn]->()
MATCH ()-[out:parWriteOut|retWriteOut]->()
WITH *, apoc.cfgPath.nonRosFindPaths(inE, {
    relSeq : "varWrite|parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : out,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"}
    ]
}) As paths
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.33: Intermediate query for Non-ROS Behaviour Alteration.

```
MATCH ()-[inE:parWriteIn|retWriteIn]->()
MATCH ()-[vif:varInfFunc]->(:cFunction)
WITH *, apoc.cfgPath.nonRosFindPaths(inE, {
    relSeq : "parWrite|retWrite*",
    filter : "cVariable,cReturn",
    endE : vif,
    shortest : true,
    config : [
        {name : "parWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn,cfgInvoke", length : "2"},
        {name : "parWrite", startLabel : "cVariable", endLabel : "cVariable",
        attribute : "cfgInvoke", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cVariable",
        attribute : "cfgReturn", length : "1"},
        {name : "retWrite", startLabel : "cReturn", endLabel : "cReturn",
        attribute : "cfgReturn", length : "1"},
        {name : "varInfFunc", startLabel : "cVariable", endLabel : "cFunction",
        length : "+"},
        {name : "varInfFunc", startLabel : "cReturn", endLabel : "cFunction",
        length : "+"}
    ]
}) As paths
UNWIND paths As path
RETURN DISTINCT path;
```

Figure C.34: Suffix query for ROS Behaviour Alteration.

```
MATCH path=
↪  (:cCompEnd)-[:dataflow|compCall|compReturn|compWrite*1..]->(:cCompStart)
RETURN DISTINCT path ORDER BY length(path) DESC;
```

Figure C.35: Full system query for Non-ROS Behaviour Alteration.