2017

Computing change of invariants to support software evolution

Ashwin Kallingal Joshy
Iowa State University

Computing change of invariants to support software evolution

by

Ashwin Kallingal Joshy

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2017

DEDICATION

I would like to dedicate this thesis to my parents whose endless support gave me the courage and motivation needed to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to all who helped me through various stages of my research. First and foremost, Dr. Wei Le for her continuous guidance, patience and support throughout the research. Her words of encouragement and trust in my abilities helped me to complete the thesis. I am grateful to my committee members, Dr. Samik Basu and Dr. Jin Tian, for their support and understanding spirit. I would also like to thank Dr. David M. Weiss for nudging me towards this research area. Lastly, I would like to thank all of my lab mates who helped to make this endeavor an enjoyable one.

ABSTRACT

Software is always evolving. In the recent years, the development community has shifted towards Agile development paradigm resulting in faster release cycle. This emphasis on speed is, generally, accompanied by an increase in the number of bugs and reduced focus on updating non-functional software artifacts like specification document. Recent studies have found that developers find it difficult to determine whether a change might break code elsewhere in the program, resulting in 25% of bugs fixes to be incorrect or buggy. A method to capture the semantic changes between different versions of a program is vital in understanding the impact of the change and in preventing bugs.

An invariant is a condition that is always true at a given program point. Invariants are used to specify the requirements and desired behavior of a program at any program point. The difference in invariants between different program versions can be used to capture the changes made to the program. In this thesis, we use the *change of invariants* as a way to capture the semantic changes over different program versions. We designed a static demand-driven algorithm for automatically computing the change of invariants between different versions of a program. To evaluate the algorithm and its ability to capture semantic changes over different program versions, we built a prototype framework called Hydrogen. Our experimental results show that Hydrogen is able to compute the change of invariants between different versions of the programs, and the computed change of invariants can be used for understanding changes and generating assertions to prevent similar bugs in future.

CHAPTER 1. INTRODUCTION

Software has become an integral part of our life, from providing entertainment to ensuring our safety and security. Any software expected to work in a real-world environment has to undergo continuous change or become progressively less useful. In order to ensure the continued relevance of their software, most developers have adopted Agile programming paradigm. While Agile development promises faster release cycles to keep the software up-to-date, the need for speed limits the time available for the developer to fully examine and understand the effect of the changes made between releases. According to Tao et al. (2012), most developers find it difficult to determine whether a change might break code elsewhere. This problem only aggravates as the software evolution speeds up; as evident from the study by Yin et al. (2011), which found that 14.8%-24.4% of bug fixes in large open source systems are either incorrect or buggy.

Design by Contract paradigm, proposed by Meyer (1992), has been shown to reduce the number of bugs. In this programming methodology, invariants in form of precondition and post-condition, are used to document the change in state caused by a piece of a program. An invariant is defined as a condition that can be relied upon to be true during execution of a program, or during some portion of it. However, a significant amount of time is required for creating and maintaining these contracts. In contrast, developers following Agile development process typically give lesser priority to non-functional software artifacts like specification document. Obsolete specifications can induce a false perception of the impact of the change, leading to bugs. Another deficiency of design by contract programming model is that it fails to take into account the evolving nature of the software.

To redress both these shortcomings, Qi et al. (2012) proposed the concept of change contracts. The underlying belief being that it is easier to specify the change in behavior between versions than to specify the absolute behavior of a program. While this approach takes into account the evolving nature of the software, it still left the daunting task of writing the change of contracts to the developers. Similarly, the concept of differential assertion, suggested by Lahiri et al. (2013), shares the belief that it is easier to document the changes than to write absolute specification. In this case, the assumption was used to speed up program verification under the premise that the developers would spend time documenting the change between versions. Rather than helping the developers understand the impact of a change, both these works, shifts the onus onto them to accurately judge its effects.

The goal of our work is to help the developers understand the impact of the changes made between different versions. Specifically, we aim to capture the semantic changes between different versions of a program. Towards that cause, we introduce the concept of *change of invariants* that can be used to comprehend the changes over different program versions. We designed a static demand-driven algorithm to automatically compute the change of invariants between different versions of a program. This helps in understanding the impact of the change rather than relying on developers' knowledge in inferring them. A framework called *Hydrogen*, consisting of three major subsystems, namely a compiler, a multi version program representation and constraint solver, was build to evaluate the algorithm.

Intuitively, change of invariants represents the change in program state or condition at a particular program point, when compared across different versions of a program. Depending on the changes made between the versions, some invariants might be added or removed at a program point. For computing the change of invariants there are two fundamental requirements. First and foremost, the program point should be shared across all the versions. This is because invariants are defined with respect to a program point and computing the change at different program points across versions won't help in understanding the semantic changes made between them. Next, we should be able to identify the program point across different versions of the program, as modifications to the program can change its location. We use a multi version program representation, introduced by Le and Pattison (2014), to help accommodate both these requirements. This representation, called Multi Version Integrated Control Flow Graph (MVICFG), also forms the base for static demand-driven algorithm for computing the change of invariants.

A naive way to compute change of invariants, would be to find out the invariants present at a program point across different versions and find the difference between them. This is not ideal for mainly two reasons. For one, this will be very expensive, especially when trying to analyze multiple version. Second, in order to find out which invariants have changed, we would require some kind of mapping between the invariants across different versions. Instead, we designed a static demand-driven algorithm that checks for invariants for all the versions simultaneously on top of MVICFG which makes identifying changed invariants a straightforward process. The MVICFG also makes it possible to share computations across different versions of a program, making the algorithm more efficient and scalable than a brute force approach.

We implemented the algorithm in a prototype framework, called Hydrogen, to evaluate it. Hydrogen is build using LLVM as the compiler back-end and Z3 as the constraint solver used for invariant detection. Experimental results from Hydrogen shows that our algorithm is capable of computing the change of invariants between different versions of a program. The computed change of invariants can be used to understand the semantic change over different program versions and in some cases be inserted as assertions to prevent similar bugs in the future.

The main contributions of this thesis are as follows:

1. The definition of change of invariant.
2. Static demand-driven algorithm to compute the change of invariants.
3. Hydrogen, a prototype framework, that implements the static demand-driven algorithm and evaluates its capability to compute the change of invariants.

The rest of the thesis is organized as follows; In Chapter 2 we provide the necessary details about the tools and concepts needed to understand the thesis followed by a summarization of the related works in this area. Chapter 3 and Chapter 4 present the concept of change of invariants and the algorithms respectively. In Chapter 5 we describe our implementation of Hydrogen and presents the experimental results gathered from it. The future work and conclusions are presented in Chapter 6.

CHAPTER 2. PREREQUISITES AND RELATED WORK

This chapter discusses the basic concepts and tools utilized in this thesis, followed by an exploration of the current frontiers in the related research areas.

2.1 Prerequisites

The main contributions of this thesis are the definition of change of invariants and an algorithm to compute them. Both of them are based on top of a multi version program representation called MVICFG. Hence, it is essential to understand the building blocks of this program representation and how it is implemented in the prototype framework, called Hydrogen. The back-end compiler and constraint solver utilized in Hydrogen are preexisting tools that have been modified to suite its needs. A basic knowledge about these tools, will help in understanding how the framework works.

2.1.1 Control flow graph

A Control Flow Graph (CFG) is a directed graph that represents all the possible execution paths that a program method can have. A CFG is build for a particular method (intra-procedural) and doesn't have edges that represent transfer of control to other methods. In this thesis, the CFG nodes represents instructions and the edges represent control flow from one instruction to another. Each CFG node has a unique ID associated with it. They also store additional information regarding the location of the instruction with respect to the program.

2.1.2 Inter-procedural control flow graph

An Inter-Procedural Control Flow Graph (ICFG) is a directed graph that combines the CFGs of all program methods. It connects a method's entry and exit sites to their call sites. A ICFG is build for an entire program and also incorporate the transfer of control between methods. In this thesis, each CFG is augmented with dummy entry and exit nodes with no program statements to ensure that every call site has single entry successor and each call site exit node has exactly one call site predecessor and one procedure exit predecessor.

2.1.3 Version control

A version Control is a system that records changes to a file or set of files over time so that you can recall specific version and see the changes made between versions.

2.1.4 Multi version inter-procedural control flow graph

The Multi Version Inter-Procedural Control Flow Graph (MVICFG) is used to represent both inter-procedural and intra-procedural changes of a program across multiple versions. Intuitively, the MVICFG is a union of the set of ICFGs across different program versions. Both the nodes and edges in MVICFG are augmented with information regarding their version. In this thesis, each node in the MVICFG also have a data structure to store the traversal and translation information required by the demand-driven algorithm.

2.1.5 LLVM

LLVM (Low Level Virtual Machine) infrastructure is a collection of modular and reusable compiler and tool-chain methods used to develop compiler front-end and back-ends. The LLVM project, started in 2000 at University of Illinois, supports compiling the source code to an Intermediate Representation (IR). LLVM IR is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing all high-level languages cleanly. In this thesis, LLVM is used as the back-end to generate ICFGs for individual versions of a program before combining them into MVICFG. To make computation of change of invariants easier, we encode the instruction in the MVICFG as LLVM IR instruction.

2.1.6 Z3

Z3 is a theorem prover from Microsoft Research. In this thesis, its state-of-the-art SMT constraint solver is used as the back-end to track, update and evaluate symbolic values when computing the change of invariants.

2.2 Related Works

The concept of invariants have been applied to software analysis for almost half a century now, starting with Hoare (1969). Invariants have been used in a wide variety of fields like Specification Mining [Ammons et al. (2002); Ramanathan et al. (2007a,b)], Path Invariants [Beyer et al. (2007)], Software Testing [Herzig et al. (2015); Sagdeo et al. (2013)], Program Refactoring [Kataoka et al. (2001); Massoni (2007)], Software Specification [Kramer and Cunningham (1979)], Assertion Verification [Lahiri et al. (2013)], Static Checking/Verification [Nimmer and Ernst (2002, 2001); Flanagan and Leino (2001); Kroening et al. (2009); Flanagan and Qadeer (2002)], Program Verification [Ghardallou (2012); Fu et al. (2008); Rodríguez-Carbonell and Kapur (2005); Kovacs and Jebelean (2005)], Fault Localization [Sagdeo et al. (2013)] and Change Contracts [Yi et al. (2015, 2013); Le et al. (2014)] being some of them. Shi et al. (2010) proposed classifying invariants into the following three categories: Characteristic-based Invariants, Relation-based Invariants, Value-based Invariants.

In spite of its application in almost all software analysis field, invariants have always been tied to a particular version of a program. Even when used in the context of evolving software like Program Refactoring and Change Contracts it has been used to specify program specification that doesn't change. To the best of our knowledge this is the first time invariants have been used to capture and represent the evolving nature of the software.

2.2.1 Invariants and assertions

Invariants and assertion are very closely related. By definition, invariants are the properties of a program at any given program point that are always true; while assertions are the properties that should always hold at any given program point according to the developer or specification. Ideally the developer's intent specified as assertion and the invariant at any given program point should be the same. Even though the importance of assertions in a project has been highlighted by Zhang and Mesbah (2015); Hatcliff et al. (2012); Shrestha and Rutherford (2011); (2002) and Casalnuovo et al. (2015) most projects are devoid of any assertion or are simple null checks as pointed out by Estler et al. (2014).

The close relation between invariants and assertion have lead to works like Vasudevan et al. (2010); Lin et al. (2013); Nimmer and Ernst (2002, 2001) and Daniel et al. (2009) that tries to deduct developer's intent from invariants, in spite of it being an unsolvable problem. The work in this thesis is similar to Nimmer and Ernst (2002) and Daniel et al. (2009) in trying to generate or update assert statements in programs but is different in two fundamental aspects.

First, our approach is not restricted to any predefined program points and hence is more fine-grained. Second, by using demand driven analysis we are able to reduce and reuse computations across versions making the amortized cost of performing the analysis significantly less. The concept of change of invariants also make it possible to extend our work, in the future, to generate differential assertions like Lahiri et al. (2013) and change contracts like Le and Pattison (2014); Yi et al. (2013) and Yi et al. (2015).

2.2.2 Syntactic differencing

In the past, some literature advocated the use of dedicated editors that kept track of the changes between versions like the works by Horwitz et al. (1989); Donzeau-Gouge et al. (1980); Notkin (1985) and Reps and Teitelbaum (1984). This fell out of favor when the extra overhead did not solve the unsolvable nature of software changes. But quite recently Muşlu et al. (2015) has revived this type of editor tracking that can help in a variety of program analysis domain.

Currently, algorithms that deducts the changes between versions without any additional meta-data are commonly used. Unix diff tries to find the longest common subsequence between two source files to match common points in them. Like Unix diff, most literature in this area like the works by Apiwattanapong et al. (2007); Raghavan et al. (2004); Yang (1991); Fluri et al. (2007) and Partush and Yahav (2014) are limited by focusing on comparing two versions of the program. History Slicing by Servant and Jones (2012) proposed a novel visualization technique that is capable of representing the entire evolution of code of interest. Similarly, the work by Nagarajan et al. (2007) can match dynamic behavior between program when matching program points using control flow graphs. Works like Kim et al. (2007) and Loh and Kim (2010) which are based on the survey conducted by Kim and Notkin (2006) can be considered the current state-of-the-art in syntactic differencing. Syntactic differencing, even though easy to compute and widely used by Version Control repositories, is of limited use to a developer when trying to analyze or understand how a program has evolved. This is where semantic differencing tools come into play.

2.2.3 Semantic differencing

Semantic differencing tries to capture the actual changes to program behavior between different versions of a program. While behavior changes have slightly different definitions based on the application, frameworks by Qi et al. (2012); Lahiri et al. (2013); Yi et al. (2013); Jackson and Ladd (1994); Kim and Notkin (2006) and Kim et al. (2007) approximates textual or specification difference to semantic change between two versions. Person et al. (2008) and Lahiri et al. (2012) takes a different approach by relying on input-output changes to appropriate semantic changes. Even though Person et al. (2008) relies on symbolic execution and Lahiri et al. (2012) on SMT solver they both attempt to compare the program behavior with different end-use applications. Similarly, the works by Buse and Weimer (2008, 2010) and Lahiri et al. (2013) focuses on reporting the semantic changes in a human readable form and tries to abstract away some irrelevant semantic changes. The work by Partush and Yahav (2014) is a state-of-the-art static semantic differencing tool that is able to handle loops by using dynamic interleaving of two programs over abstract domain.

The work in this thesis, in regard to semantic differencing, is a re-implementation of MVICFG by Le and Pattison (2014) using LLVM IR as base. Hence, it is more similar to Horwitz et al. (1989); Horwitz (1990) and Raghavan et al. (2004) due to its dependency on graph but is fundamentally different from all other work in one key aspect. While vast majority of semantic differencing tools work on two versions, MVICFG is able to handle multiple versions and characterize their semantic differences.

Programming analysis have typically focused on latitudinal exploration of a program, i.e determining facts about a program point within that instance of the program. Longitudinal program analysis, i.e analysis of a program point across the multitude of versions created during a program's lifetime, and its advantages were highlighted by Notkin (2002). When demand driven analysis is carried out on top of Hydrogen, our approach is taking maximum advantage of longitudinal program analysis.

2.2.4 Dynamic invariant generation

Invariant detection and generation can be broadly classified into dynamic and static generation techniques. While dynamic invariant generation techniques often offer more concrete and stronger invariants over their static counterparts as explained by Ernst (2004), they can be unsound and easily mislead by test suite artifacts. This is illustrated in this work by Ernst et al. (2007). Static inference techniques, on the other hand, are sound (at least theoretically) but are very conservative. In practice, both techniques have some unsoundness in them as a result of trade-off between scalability, run-time constrains and lack of source code for library and API calls.

Daikon by Ernst et al. (1999) is one of the most notable work done in generic dynamic invariant generation. It has been further advanced in works by Ernst et al. (2001) and Perkins and Ernst (2004). Daikon uses execution traces obtained from running instrumented version of the program to infer invariants from predefined templates. Some dynamic invariant generation tools are designed for specialized purposes like the work by Colón et al. (2003) which generates liner invariants with the help of constraint solving, Csallner and Smaragdakis (2006) that mines interface invariants dynamically, Nguyen et al. (2012) that specializes in polynomial and array based invariants, Shi et al. (2010) which computes invariants that define definition-usage patterns, Sagdeo et al. (2011) which infers invariants at function and loop boundaries through program path guided clustering, Yang et al. (2006); Nguyen et al. (2014) and Xie and Pei (2006) that mines temporal rules for APIs.

DIDUCE by Hangal and Lam (2002) is a lazy dynamic invariant generator that starts by establishing the strongest possible template and relaxing them as violations are found. DySy by Csallner et al. (2008) improves relevance of dynamically inferred invariants, or reduce the size of the test suites required by combining concrete execution with symbolic execution and can be considered as a notable improvement to current state-of-the-art in generic dynamic invariant generation. While dynamic invariant generators are generally not good in handling multi-threaded program, the work by Kusano et al. (2015) has modified the instrumentation process of Daikon to make it the current state-of-the-art tool in multi-threaded dynamic invariant generation.

2.2.5 Static invariant generation

The main advantage of using static techniques for invariant generation is that the reported invariants are true for every reachable state of the program. The work by Gupta and Rybalchenko (2009), which uses constrains solving and that by Dillig et al. (2013), which relies on abducting inference to generate invariants to assist in program verification are among the state-of-the-art works tools for static invariant generation. The work by Sankaranarayanan et al. (2004) that infers non-linear loop invariants by using theory of ideals over polynomial rings and the those by Colón et al. (2003); Bozga et al. (2009); Bradley et al. (2006) and Henzinger et al. (2010) that generates invariants related to integer arrays are a few specialized invariant inference tools.

But static analysis in itself finds plenty of application in static verification tools and model checking tools like in the works of Nimmer and Ernst (2002, 2001); Flanagan and Leino (2001); Kroening et al. (2009); Flanagan and Qadeer (2002); Ghardallou (2012); Fu et al. (2008); Rodríguez-Carbonell and Kapur (2005) and Kovacs and Jebelean (2005). Static analysis or Model checking tools typically relies on symbolic execution like Csallner et al. (2008); Schmitt and Weiß (2007) and Păsăreanu and Visser (2004) or abstract interpretation like those by Cousot and Cousot (1977a,b); Cousot and Halbwachs (1978); Miné (2006) and Laviron and Logozzo (2009) or symbolic model checking to infer conditions or invariants. While symbolic execution and model checking methodology suffers from scalability and state explosion issues, abstract interpretation relies on over-approximation that can lead to loss of some information.

The work in this thesis is more closely related to the work by Gupta and Rybalchenko (2009), in its reliance on constraint solving and invariant templates, but is novel in its demand-driven propagation mechanism that enables reuse and reduces computation across different versions of a program.

CHAPTER 3. DEFINING CHANGE OF INVARIANT

In this chapter, we will explore and define the concept of change of invariants with the help of examples. But before we can get to change of invariants, we need to disambiguate the basic terms that will be used in defining it, namely *program point* and *matched program point*.

3.1 Prerequisites

One of the fundamental requirement for computing the change of invariants is that the program point being analyzed should be shared between the different versions of the program. The program point, without context, is a very ambiguous term. For example, when we are talking about a program point in the context of grammar of the program, it can take the definition of an atomic statement in the program. Similarly, when talking in the context of the source code of a program, it can take the form of individual lines irrespective of the number of statements present in it. Since our static demand-driven algorithm relies on the multi version program representation called MVICFG, in this thesis, we will be using it as the context for defining a program point. Based on this, we can define a program point as follows.

Definition 1 *A* ***program point*** *is a node in MVICFG.*

MVICFG captures both the inter-procedural and intra-procedural changes across different versions of a program. The edges in MVICFG are annotated with version

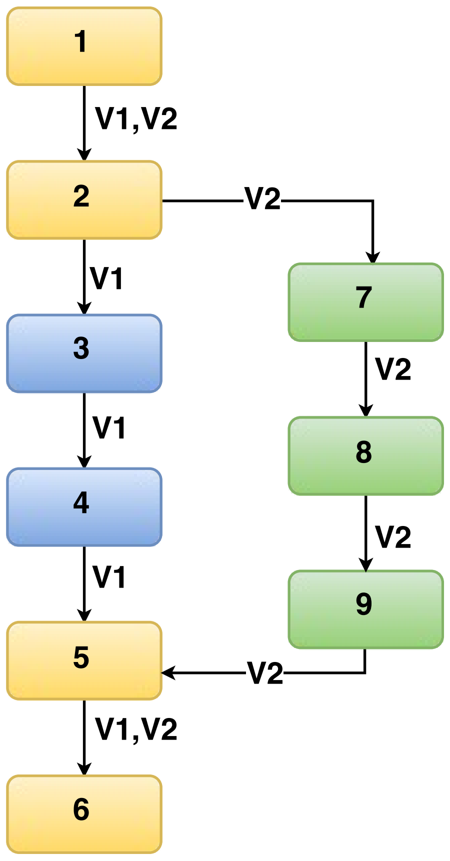


Figure 3.1: MVICFG for two versions to show matched points

information to reflect the changes occurring over the program versions. Intuitively, for a program point to be matched across different versions of a program it has to be present across the different program versions. Extending this intuition, we can define a matched program point as follows.

Definition 2 *A **matched program point** is a `program point` in the MVICFG, that is shared across the program versions.*

Consider a two version MVICFG as shown in Figure 3.1. For the sake of simplicity, only inter-version difference is expressed in this MVICFG. As seen in the figure, for creating the second version the blue nodes (3 and 4) were replaced with green nodes (7-9). The edges from node 2 and those going to node 5, encapsulates the changes made between the two version of the program, with the help of the annotations. Intuitively, we can see that the nodes in orange nodes (1, 2, 5 and 6) are common across the two versions of the program and hence, can be matched between them. Hence, according to the definition, the nodes 1, 2, 5, 6 are matched program points for the MVICFG shown in Figure 3.1.

3.2 Change Of Invariant

Now that we have a precise way to define program points, let's explore the concept of change of invariants. Intuitively, a change of invariant portray a change in program state or condition at a program point when compared across different versions of a program. Accordingly, we can define change of invariants as follows:

Definition 3 *The **change of invariants** is a 4 tuple `(MP, V1, Type, V2)`; where MP is the `matched program point`, `V1` is the invariant expressed in conjunctive normal form for the first version(s) of the program, `Type` specifies the update kind in terms of addition, modification and deletion, and `V2` is the invariant expressed in conjunctive normal form for the second version(s) of the program.*

Consider a function to convert the temperature from Fahrenheit to Celsius as shown in Figure 3.2.

```
double conv(double f) {
    double c = (f - 32.0) * 5.0 / 9.0;
    return c;
}
```

Figure 3.2: Function to converts temperature from Fahrenheit to Celsius; Version 1

Let's suppose during refactoring, the developer, removed the parenthesis from the formula as shown in Figure 3.3.

```
double conv(double f) {
    double c = f - 32.0 * 5.0 / 9.0;
    return c;
}
```

Figure 3.3: Function to converts temperature from Fahrenheit to Celsius; Version 2

The MVICFG over the two versions is show in Figure 3.4. It follows a similar color code as the MVICFG in Figure 3.1. The unique number associated with the nodes in the MVICFG is shown on the top left corner. Nodes 1 and 4 are special nodes in MVICFG used to show the entry and exit of a function. From the definitions above, we can see that node 3 is one of the matched program point. At node 3, if we calculate the invariant for the function in Figure 3.2, we get `[c = (f - 32) *(0.555555556)]`. On the other hand, the invariant reported at node 3 for the function in Figure 3.3 would be `[c = f - 17.777777778]`, i.e. The relation between the variables present at the matched program point got modified. Hence, the change of invariants at matched program point would be `(3, [c = (f - 32) *(0.555555556)], Modify, [c = f - 17.777777778])`.

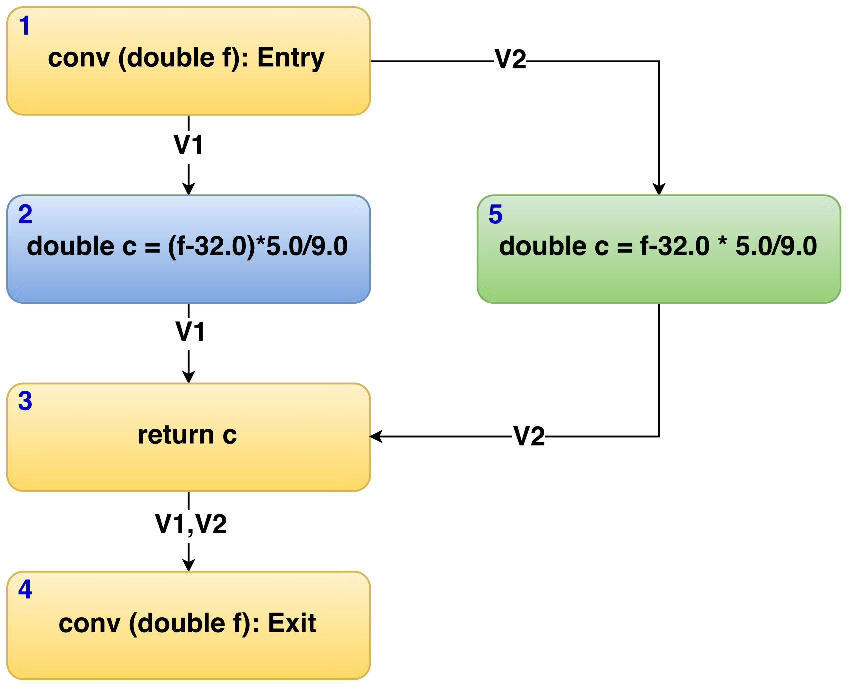


Figure 3.4: MVICFG for two versions of a function to convert temperature from Fahrenheit to Celsius

Now let's consider the program versions represented by the MVICFG shown in Figure 3.5. In the first version, the developer forgot to input the value for one of the variables, which is rectified in the second version by the addition of node 5. If we calculate the invariants present at the matched program point 3, for the first version we get `[b = a + 10]`. The invariant at node 3 for the second version would be `[a = Input ∧ b = a + 10]`. As a new relation is formed between the variables, in this example, the change of invariants at the matched program point would be `(3, [b = a + 10], Add, [b = a + 10 ∧ a = Input])`.

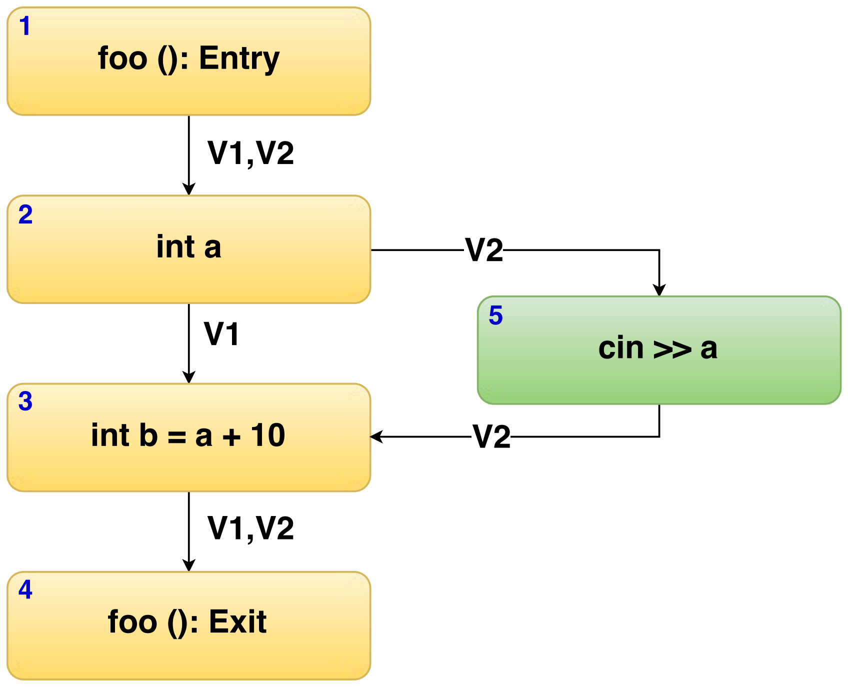


Figure 3.5: MVICFG for illustrating addition type of change of invariants

With the concept of change of invariants clearly defined, the next chapter presents an efficient algorithm to compute the same.

CHAPTER 4. COMPUTING CHANGE OF INVARIANT

This chapter presents the static demand-driven algorithm that can be used to compute the change of invariants between different versions of a program. In Section 4.1 we present an overview of the Hydrogen framework, which encapsulates all the algorithms discussed here. Since the definition and computation of change of invariants are based on top of MVICFG, Section 4.2 will focus on generation an instruction level MVICFG. Section 4.3 will present the static demand-driven algorithm implemented on top of the MVICFG.

4.1 Overview

Algorithm 1 presents an overview of the Hydrogen framework, that encompasses all the other algorithms. It takes different versions of the program, a program point of interest and the program version(s) of interest as the input. The output is the computed change of invariant at the specified program point over the interested program versions.

Algorithm 1 Overview of Hydrogen Framework

```
1: Input: Program versions[$V_1$,$V_2$,...,$V_n$], Program Point (PP), Interested versions(V)
2: Output: Change of Invariant at $PP$ over $V$
3: function HYDROGEN(Program versions, Program Point, Interested versions)
4:     MVICFG ← GENERATEMVICFG([$V_1$,$V_2$,...,$V_n$])
5:     if PP is a matched program point then
6:         COMPUTECHANGEINVARIANT(MVICFG, PP, V)
7:     end if
8: end function
```

The MVICFG for all the input program is generated at line 4 as the first step. At line 5, we check the given program point over the interested program versions to make sure that it is common to all of them. This ensures that the given program point is a shared program point as defined in Definition 2. Once we confirm that it is a shared program point, we call the demand-driven algorithm to compute the change of invariants. At the end of the function call, it outputs the computed change of invariants.

Now that we have an overview of the process, let's consider the generation of MVICFG next.

4.2 MVICFG Generation

In this section we will explore the algorithms needed to generate the MVICFG.

Algorithm 2 Algorithm to Generate MVICFG

```
1: Input: Program versions[$V_1$,$V_2$,...,$V_n$]
2: Output: MVICFG
3: function GENERATE GRAPH(Program versions)
4:     MVICFG ← BUILDICFG($V_1$)
5:     for all $V_i \in [V_2, V_3, \ldots, V_n]$ do
6:         ICFG ← BUILDICFG($V_i$)
7:         D ← GENERATEDIFF($V_i$, $V_{i-1}$)
8:         for all diff ∈ D do
9:             if diff = Add then
10:                N ← IDENTIFYADDEDNODES(ICFG, diff)
11:                ADDNODES(MVICFG, N)
12:                NewNodes ← N
13:            else if diff = Del then
14:                N ← IDENTIFYDELETEDNODES(ICFG, diff)
15:                DelNodes ← N
16:            end if
17:        end for
18:        ADDTOMVICFG(NewNodes, ICFG, MVICFG, diff)
19:        DELETEFROMMVICFG(DelNodes, ICFG, MVICFG, diff)
20:        UPDATEVERSION(MVICFG, D)
21:    end for
22: end function
```

Le and Pattison (2014) originally proposed the algorithms to generate MVICFG. The algorithms presented here are only trivially different. Algorithm 2 outlines the basic procedure for generating an MVICFG. It takes as input, the different versions of the program and generates the MVICFG as its output.

The Algorithm 2, can be broadly split into three parts. In the first part we build the ICFG of the initial version of the program to be used as the base for building the MVICFG. The second part deals with identifying the changes that happened across the program versions. Lastly, we update the edges in MVICFG to reflect the control flow from all the versions and update the version information embedded in the nodes and edges.

More specifically, at line 4, the ICFG of the initial version is generated. Lines 5-20 incrementally integrates the differences in program versions, one version at a time. At line 6, we build the ICFG of the next version to be integrated. We then obtain, the statement level differences between the two versions using UNIX **diff** tool at line 7. The diff generator reports only addition and deletion of statements. The changed statements are reported as deleted from the old version and added in the new version. Lines 8-17 process each diff line generated in line 7 to identify nodes to be added or deleted from the MVICFG. If statements are added, we identify the nodes to be added from the ICFG at line 10 and adds them to MVICFG in line 11. A set called `New Nodes` is updated at line 12 to track all the nodes added to the MVICFG. On the other hand if statements are deleted, we identify the nodes to be removed from the MVICFG at line 14 and track them in the set `Del Nodes` at line 15.

To help us understand these steps better, let's try applying them to an example. In the interest of keeping things simple, let's consider the function shown in Figure 4.1a. Let's suppose that in the second version, we removed the `else` branch and decided to add the statement `b--` in the `if` branch as shown in Figure 4.1b. The corresponding control flow graphs of the two versions are shown in Figure 4.2. Let's use this example

```
void foo() {
    int a = 0;
    int b = 1;
    if (a == 0) {
        b++;
    } else {
        b--;
    }
    a ++;
    cout<< b;
}
```

(a) Version 1

```
void foo() {
    int a = 0;
    int b = 1;
    if (a == 0) {
        b++;
        b--;
    }

    a ++;
    cout<< b;
}
```

(b) Version 2

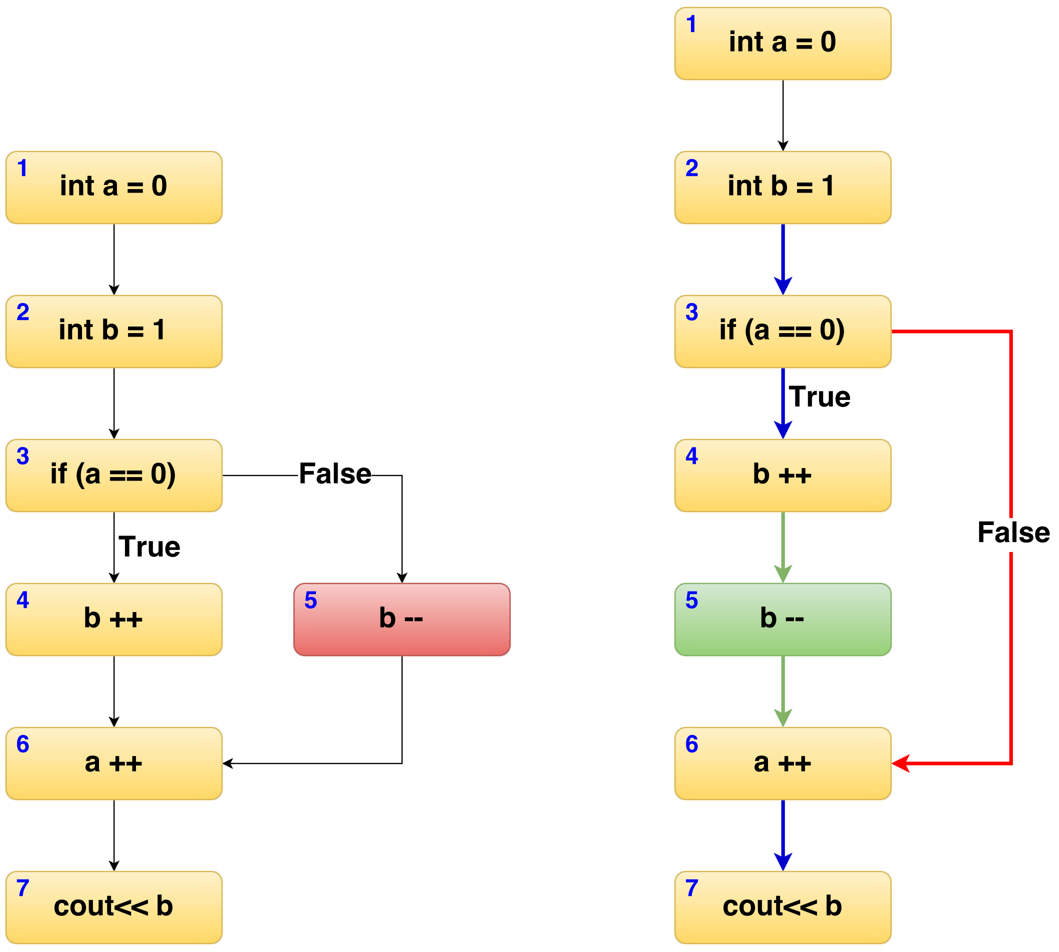
Figure 4.1: Program versions to demonstrate generation of MVICFG

to work through the algorithm. The significance of the colored nodes and edges will be demystified shortly.

In Figure 4.2, we have highlighted the additions and deletions, happening between the two versions, by marking the nodes with green and red color, respectively. At line 11 of Algorithm 2, the green node from Figure 4.2b would be added to the MVICFG. Similarly, at line 15, we would mark the node matching the red node in Figure 4.2a in the MVICFG. Hence, at line 17, the in-progress MVICFG would be as shown in Figure 4.3.

Once we have identifying all the added and deleted nodes, we have to update the edges in MVICFG. At line 18, we add the necessary edges to connect the added nodes, present in the set `New Nodes`, to MVICFG through Algorithm 3. Similarly, Algorithm 4 updates the edges for all the deleted nodes present in `Del Nodes`. At line 20, we update the version information associated with the edges and nodes in the MVICFG by using Algorithm 5.

Algorithm 3 details the process to connect the newly added ICFG nodes to MVICFG. The goal is to identify the control flow for the newly added nodes and append it to



(a) Version 1 (b) Version 2

Figure 4.2: ICFGs of programs shown in Fig. 4.1

MVICFG. Lines 2-8 appends the control flow for each added node individually. To connect the node to its predecessors and successors in the MVICFG, we first identify its predecessors and successors in the ICFG (line 3). We then map these entries and exits of the differences to MVICFG in lines 4-6. This appends the missing control flow from the added nodes to MVICFG.

Going back to the example in hand, there is only one predecessor (node 4) for the added green node in Figure 4.2b. Since the corresponding node in the in-progress MVICFG is 4, as shown in Figure 4.3, we add an edge connecting node 4 to 8. Sim-

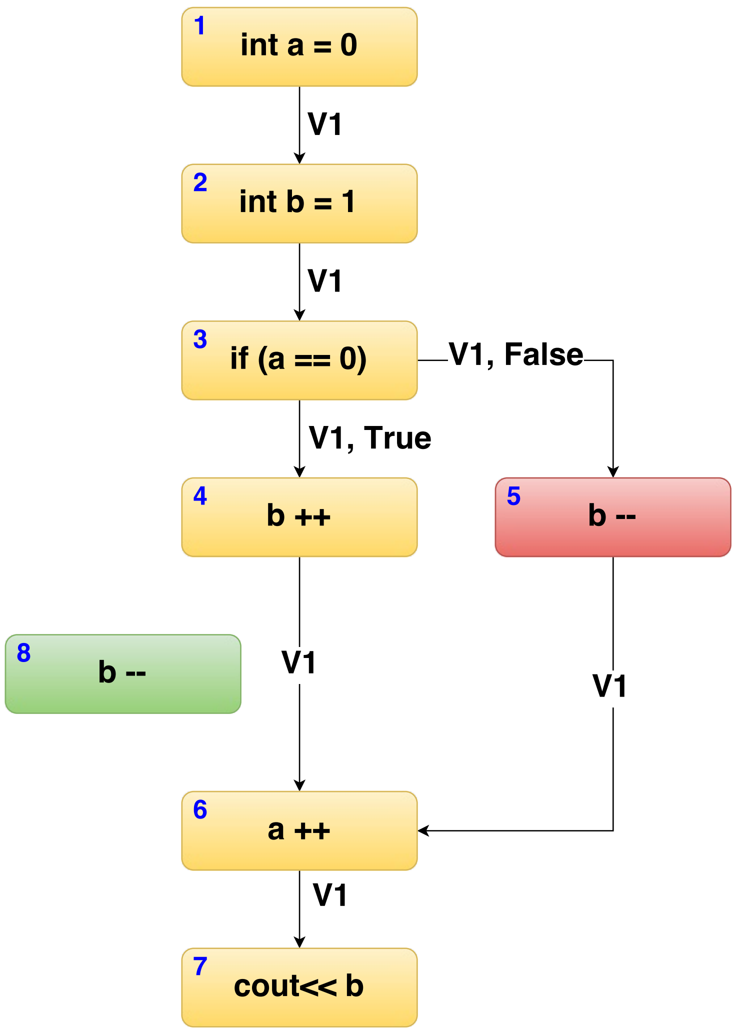


Figure 4.3: In-progress MVICFG at line 17 of Algorithm 2

ilarly, for the one successor, node 6 in Figure 4.2b, we add the edge connecting the green node 8 to the corresponding matched node in MVICFG. Hence, at line 18 of Algorithm 2, the in-progress MVICFG would be as shown in Figure 4.4. The edges added to append the control flow of added nodes have been highlighted in green in both Figure 4.2b and Figure 4.4.

The deleted nodes in MVICFG are not removed. Instead, the edges are updated to reflect the change in control flow as shown in Algorithm 4. Lines 2-15 update the edges for each deleted nodes individually. We find the predecessors and successors of

Algorithm 3 Algorithm to connect newly added ICFG nodes in MVICFG

```
1: function AddToMVICFG(NewNodes, ICFG, MVICFG, diff)
2:     for all N $\in$ NewNodes do
3:         ConnectedNodes $\leftarrow$ Pred(N, ICFG) $\cup$ Succ(N, ICFG)
4:         for all C $\in$ ConnectedNodes AND C $\notin$ NewNodes do
5:             C' = FindMatchedNode(C, MVICFG)
6:             AddEdge(C', N)
7:         end for
8:     end for
9: end function
```

Algorithm 4 Algorithm to update edges for deleted ICFG nodes in MVICFG

```
1: function DeleteFromMVICFG(DelNodes, ICFG, MVICFG, diff)
2:     for all N $\notin$ DelNodes AND N $\in$ MVICFG do
3:         ConnectedNodes $\leftarrow$ Pred(N, ICFG) $\cup$ Succ(N, ICFG)
4:         for all C $\in$ ConnectedNodes do
5:             if C $\in$ DelNodes then
6:                 N' $\leftarrow$ FindMatchedNode(N, ICFG)
7:                 for all M' $\in$ Pred(N', ICFG) $\cup$ Succ(N', ICFG) do
8:                     M $\leftarrow$ FindMatchedNode(M', MVICFG)
9:                     if NoEdge(M, N) then
10:                        AddEdge(M, N)
11:                    end if
12:                end for
13:            end if
14:        end for
15:    end for
16: end function
```

the deleted node in MVICFG and map them to ICFG in lines 4-6. We then check all the edges from this mapped ICFG node, to see if there are any missing edges in MVICFG in lines 7-9. In case a missing edge is identified, we add it to the MVICFG at line 10.

In case of our example, the algorithm finds the predecessors and successors of the deleted red node from the MVICFG (nodes 3 and 6 in Figure 4.4). Then, from the ICFG of the second version, we find all the edges associated with them to see if any of the edges are missing from the in-progress MVICFG. These edges have been highlighted in blue in Figure 4.2b except for the edge connecting node 3 to node 6. The edge (3,6) is missing from the in-progress MVICFG and has been highlighted in red.

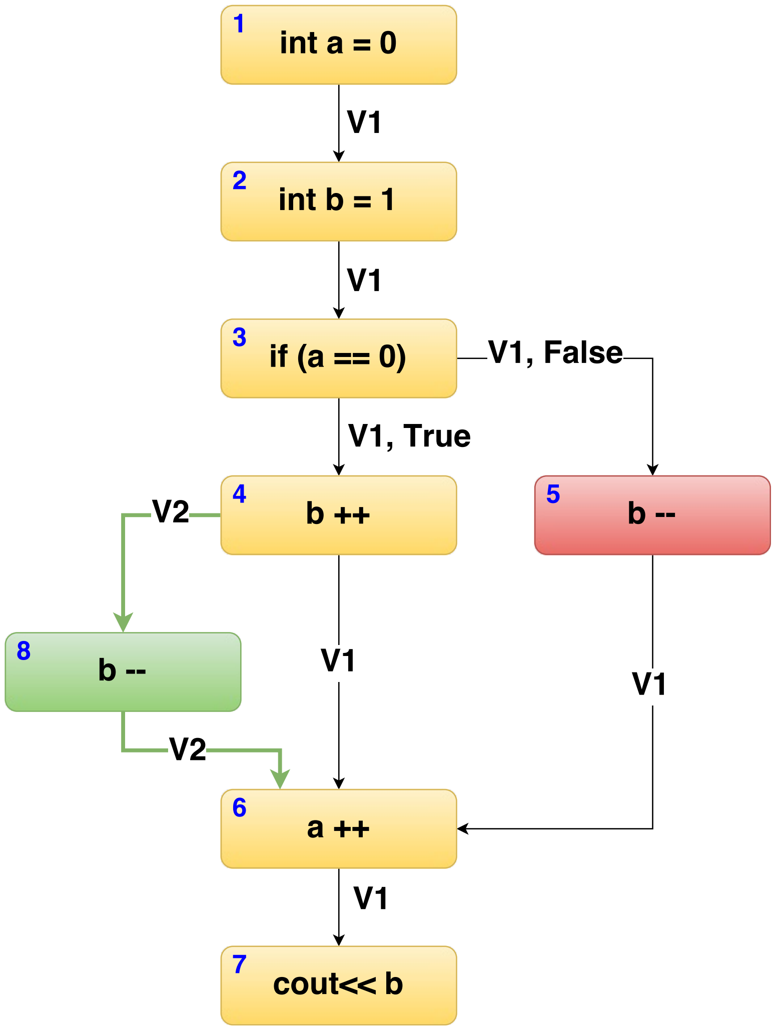


Figure 4.4: In-progress MVICFG at line 18 of Algorithm 2

Once the missing edges have been identified, we add them into the in-progress MVICFG. Hence, at line 19 of Algorithm 2, the in-progress MVICFG would be as shown in Figure 4.5.

Algorithm 5 updates the version information for the edges and update the nodes' location information to reflect the current ICFG versions. This ensures that the loop at lines 8-17 of Algorithm 2 can integrate the next ICFG. Lines 2-10 updates each node's location information individually. If the node was not added or deleted then we update its location information from diff information in lines 3-5. On the other hand, if it is a

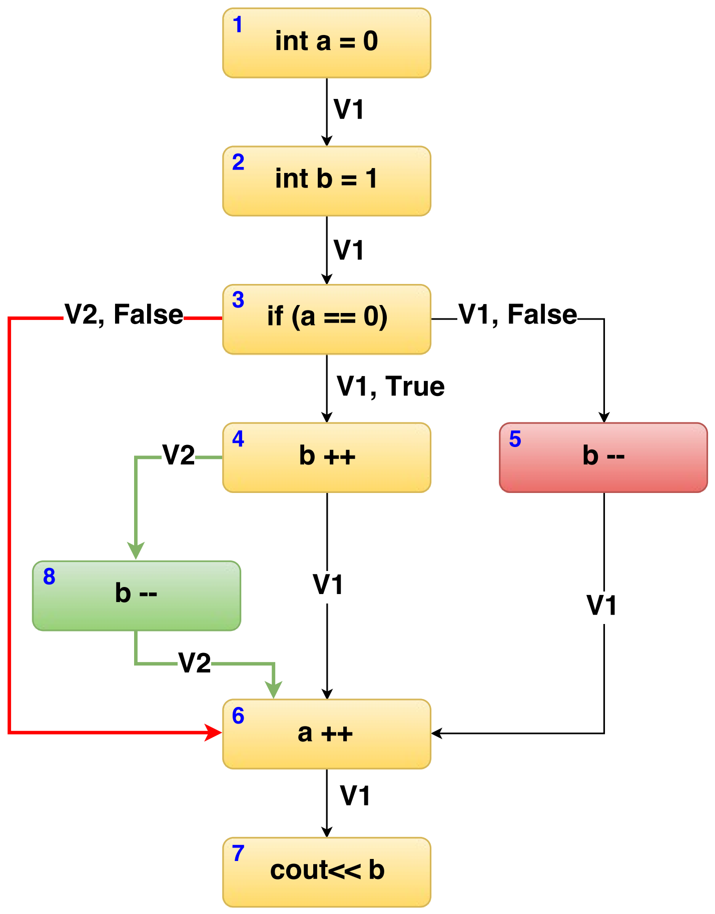


Figure 4.5: In-progress MVICFG at line 19 of Algorithm 2

deleted node, then we set its location to be empty in lines 4-9. Similarly, lines 11-17 updates the edges in the MVICFG. If the edge connects either to or from a deleted node, we leave the edge unchanged. If it doesn't connect a deleted node and the starting node doesn't have any new outgoing edge in the ICFG, we update the edge's version to reflect that it is shared between the two versions in line 12-16.

In our example, the edges (1,2), (2,3), (3,4), (4,6), (6,7) do not connect to a deleted node. Then we check to see if the starting nodes of these edge have any new outgoing edge in the second ICFG. In this example, for the edge (4,6), the starting node 4 has a new outgoing edge. This edge is removed from the list of edges to be

Algorithm 5 Algorithm to update the version tag(s) for Edges and line number for Nodes

```
1:  function UPDATEVERSION(MVICFG, NewNodes, DelNodes, ICFG, D)
2:      for all N $\in$ MVICFG do
3:          if N $\notin$ DelNodes then
4:              if N $\notin$ NewNodes then
5:                  Update node's location using info in D
6:              end if
7:          else
8:              Set node's location to empty
9:          end if
10:     end for
11:     for all E $\in$ MVICFG do
12:         if E.ToNode $\notin$ DelNodes then
13:             if E.FromNode $\notin$ DelNodes then
14:                 if E.FromNode has no new outgoing edges in ICFG then
15:                     Add version of ICFG to edge
16:                 end if
17:             end if
18:         end if
19:     end for
20: end function
```

updated. For the rest of the identified edges, we append the new version's information. At line 20 of Algorithm 2, the generated MVICFG is shown in Figure 4.6.

Now that we can generate an MVICFG, let's see how to compute the change of invariants on top of it.

4.3 Computing The Change Of Invariants

Algorithm 6 is one of the main contributions of this thesis. It takes as input, an MVICFG, the matched program point and interested version(s) over which to compute the change of invariant. The output of the algorithm is change of invariant computed for the matched program point over the specified version(s).

We extract the node associated with shared program point from the MVICFG at line 6. At line 7, two things happens. First, we raise the query at the node over the required

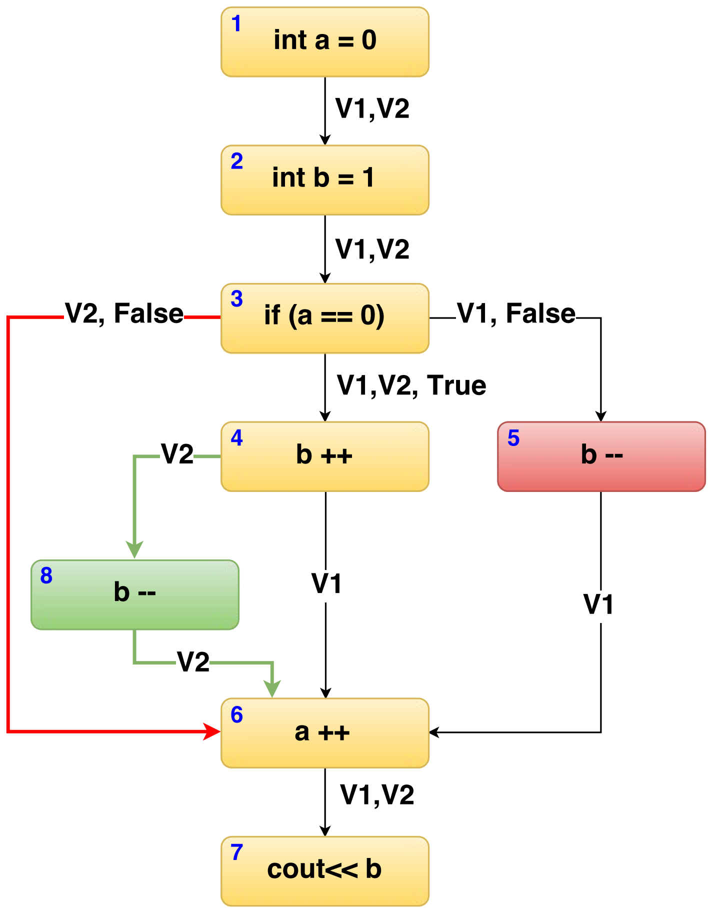


Figure 4.6: Final MVICFG at line 20 of Algorithm 2

program versions using Algorithm 7. Then we insert this query into the work-list. The lines 8-15 loop, as long as the work-list is not empty. Inside the loop, we extract the top query from the work-list at line 9. Then, at line 11, we try to resolve the query using Algorithm 8. If the query was not resolved then we propagate the query using Algorithm 9 at line 13. On the other hand, if the query is resolved then there is no need to propagate it any further. When work-list is empty, we output the change of invariant using the Algorithm 12 at line 16.

To understand the algorithms better, let's apply it to the MVICFG generated in Section 4.2. At an intuitive level, Algorithm 6 raises a query at the program point of in-

Algorithm 6 Algorithm to compute the change of invariants between versions

```
 1: Input: MVICFG (MVICFG), Program point (PP), Interested versions (V)
 2: Output: Change of Invariant at Program Point (PP)
 3: function ComputeChangeInvariant(MVICFG, PP, V)
 4:     ProcessedNodes ← $\emptyset$
 5:     Resolved ← False
 6:     N ← MVICFG.getNode($PP$)
 7:     Worklist ← RaiseQuery(N, V)
 8:     while Worklist $\neq \emptyset$ do
 9:         Remove a query Q from the front of the Worklist
10:         CurrentSymbolicValue ← $\emptyset$
11:         Resolved ← ResolveQuery(Q, &CurrentSymbolicValue)
12:         if Resolved $\neq$ True then
13:             PropagateQuery(MVICFG,Q,ProcessedNodes,CurrentSymbolicValue)
14:         end if
15:     end while
16:     OutputChangeOfInvariant(V)
17: end function
```

terest and tries to resolve the query by propagating it backwards through the MVICFG. The propagation stops when the query is resolved, in conflict with another query or there are no more nodes left to propagate the query to. When all the queries have been resolved, we compute the change of invariants over the interested versions by identifying which invariants are not shared across the versions of interest.

Algorithm 7 Algorithm to Raise a query at node N

```
 1: function RaiseQuery(N, V)
 2:     Q.Node ← N
 3:     Q.Version ← V
 4:     LiveVars ← GetLiveVariables(N, MVICFG)
 5:     for all var $\in$ LiveVars do
 6:         Q.Variable ← var
 7:         for all template $\in$ InvariantTemplates do
 8:             Q.Template ← template
 9:             Add the query Q to Worklist
10:         end for
11:     end for
12:     return Worklist
13: end function
```

A query raised in Algorithm 7 consists of the following attributes; the node at which the query is currently, the version(s) over which the query is to be evaluated, the variable being evaluated and the invariant template used to evaluate the query. The first two attributes are the same as the input to Algorithm 6. Since the change of invariants can be caused by a variable that is explicitly not present in the instruction at the node, we gather all the live variables at that node (line 4). Then in lines 5-11, we handle each live variable individually. For each variable and invariant template pair, we add to work-list a query at line 9. Once all the queries have been raised, the work-list is returned at line 12.

For the sake of simplicity in the example, let's assume that the query we want to raise at node 7 of Figure 4.6, only seeks to answer whether the variable `a`, satisfies the template `x = C`, where `x` is any variable and `C` is any constant. Hence, Algorithm 7 will generate and insert into the work-list, at line 7 of Algorithm 6, the query shown in Figure 4.7.

```
Q.Node     :  7
Q.Version  :  V1,V2
Q.Variable :  a
Q.Template :  x = C
```

Figure 4.7: Sample query generated by Algorithm 7

Algorithm 8 updates the symbolic value of the variable under evaluation and tries to solve the invariant template. If the constraint solver is able to resolve the query, at line 6, then we use Algorithm 11 to track the resolved query as a possible invariant at line 7. If the query is not resolved, then there is no invariant candidate to track. At line 10, we add the details about the query and the current symbolic value of the query variable to a data structure called `Processed Nodes`, to keep track of the query and the nodes it visited so far. This help in Algorithm 9 to terminate some queries early, adding to scalability and efficiency of the algorithm. We return whether the query was resolved or not, in line 12. Any constraint solver with support for symbolically updating the variable under evaluation and the ability to check if a particular template is true, can be used here.

Algorithm 8 Algorithm to update the symbolic value in order to the resolve the Query

```
1: function ResolveQuery(Q, &CurrentSymbolicValue)
2:     Resolved ← False
3:     template ← Q.Template
4:     versions ← Q.Version
5:     UpdatedSymbolicValue ← $\emptyset$
6:     if ConstraintSolver(Q, template, &UpdatedSymbolicValue) then
7:         UpdateInvariants(UpdatedSymbolicValue, Versions)
8:         Resolved ← True
9:     end if
10:    Add pair (Q, UpdatedSymbolicValue) to ProcessedNodes
11:    CurrentSymbolicValue ← UpdatedSymbolicValue
12:    return Resolved
13: end function
```

Going back to our example, we propagate the query shown in Figure 4.7 backwards through the MVICFG until it is resolved or terminated (loop in the lines 8-15 of Algorithm 6). In our case, during the first iteration when we try to resolve the query at line 11 using Algorithm 8, it would return that the query cannot be resolved at node 7 of Figure 4.6. Since the instruction at node 7 doesn't alter the query variable `a`, no symbolic change, with respect to the query occur in this iteration. Hence, other than adding the query and the node to the set `Processed Nodes`, no other changes happen.

In Algorithm 9, we propagate the query in case Algorithm 8 was not able to resolve it at the current node. We find the predecessors of the current node along any edge whose version have at least some version in common with query's version (line 2-5). If the node has not been previously visited by the same query or if the symbolic value from that visit matches the current symbolic value for the query variable, we use Algorithm 10 to update the query at lines 7 and 13, respectively. In case the node was

Algorithm 9 Algorithm to propagate the Query through MVICFG

```
1: function PropagateQuery(MVICFG, Q, CurrentSymbolicValue)
2:     for all E $\in$ MVICFG.Edges do
3:         Intersection $\leftarrow$Intersect(E.Version, Q.Version)
4:         if Intersection $\neq \emptyset$ then
5:             if E.ToNode = Q.Node then
6:                 if (E.FromNode, Intersection) $\notin$ ProcessedNodes then
7:                     UpdQuery(E.FromNode, Intersection, Q.Template, Q.Variable)
8:                 else
9:                     cachedValue $\leftarrow$ ProcessedNodes(Q).UpdatedSymbolicValue
10:                    if cachedValue $\neq$ CurrentSymbolicValue then
11:                        RemoveInvariant(Q, Intersection)
12:                    else
13:                        UpdQuery(E.FromNode,Intersection,Q.Template,Q.Variable)
14:                    end if
15:                end if
16:            end if
17:        end if
18:    end for
19: end function
```

visited by the same query carrying a different symbolic value for the query variable, we use Algorithm 13 to remove any conflicting candidate invariants.

In the example, the query wasn't resolved at node 7 of Figure 4.6. Hence, we propagate it backwards using Algorithm 9. In doing so we observe that there is an edge that shares the same versions as the query's version, namely the edge (6,7). Since we don't have any other information from other queries at this point and we update the query at line 7 of Algorithm 9 using Algorithm 10.

Algorithm 10 Algorithm to Update the query at node N

```
1: function UpdQuery(N, Version, Template, Variable)
2:     Q.Node $\leftarrow$ N
3:     Q.Version $\leftarrow$ Version
4:     Q.Variable $\leftarrow$ Variable
5:     Q.Template $\leftarrow$ Template
6:     Add the query Q to Worklist
7:     return Worklist
8: end function
```

Algorithm 10 updates the query. When updating the query, we don't modify the query variable or the invariant template. This is kept constant by Algorithm 9. The query node, get updated to reflect the current node being evaluated and the query version is set to be intersection of query's version and the version of edge it traversed to reach the node. At line 6, we add this updated query into the work-list and return the work-list at line 7.

```
Q.Node      :  6
Q.Version   :  V1,V2
Q.Variable  :  a
Q.Template  :  x = C
```

Figure 4.8: Sample query after it is updated by Algorithm 10 at node 6

In our example, when the query is updated after propagating it to node 6 in Figure 4.6, only the `Node` attribute will be changed as shown in Figure 4.8. In the second iteration of the loop in lines 8-15 of Algorithm 6, we try to resolve the query at node 6. Here, however, the instruction `a++` is present. Therefore, we update the symbolic value of the query variable to reflect this and store the information that a query for checking whether '`a = C`' has passed through this node and its symbolic value was [a_2 = + 1 a_1]. The state of the algorithm after this iteration is shown in Figure 4.9. The nodes visited by the query have been colored blue for easier identification.

Continuing with the example, when Algorithm 9 is called to propagate the query from node 6, there are multiple possible paths with different edge versions. Hence, the query get split at this point and we will use different colors for the edges and the nodes to capture where the symbolic update at node came from. Different edges connect the node 6 to nodes 3, 8, 4 and 5, but none of the nodes have instruction that affect the query variable '`a`'. However, when updating the query using Algorithm 10, the query version has to be adjusted according to the edge it traversed to reach the current

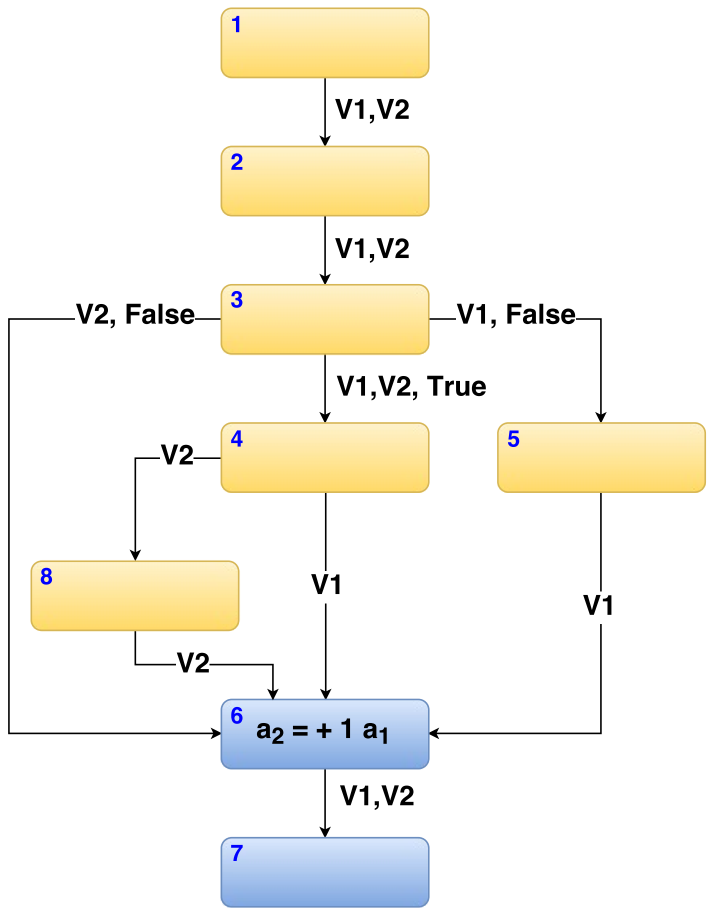


Figure 4.9: Query state; Snapshot 1

node. For example, when the node propagated from node 6 to 8, the Algorithm 10 will update the query as shown in Figure 4.10.

Again, as none of the nodes in question were visited before, the query state after propagation is as shown in Figure 4.11. The version for which the symbolic value is valid is shown in the bracket inside the node whenever the value is not common to both the versions. At the next iteration, when the query at node 8 (red) propagates to node 4 (gray), it detects that the node has been visited by a query asking the same question. But since the symbolic value present at node 4 was made exclusively for version 1, no comparison is made between the queries and the query is propagated

```
Q.Node     : 8
Q.Version  : V2
Q.Variable : a
Q.Template : x = C
```

Figure 4.10: Sample query after it is updated by Algorithm 10 at node 8

further. In case of the queries being propagated from nodes 4 and 5, it detects that the preceding nodes were already visited by a query asking the same question and that the symbolic value is shared with the queries' version. However, as no conflict is found between the symbolic values being propagated by the different queries, the propagation carries forward. This continues till orange query reaches node 1. At this point, the query can be successfully resolved and the Algorithm 11 is called to add this as a possible invariant candidate.

Algorithm 11 is used to update the invariant candidates during the computation of change of invariants. In case no candidate invariants were present, the input candidate is added to the list of invariant candidates at line 4. On the other hand, if there are preexisting invariant candidates, then lines 8 and 9 check if there is an invariant candidate for the same variable and template. In case of such an invariant candidate, line 11 sets the Boolean variable `invExist` to true, if the preexisting invariant candidate's and input invariant candidate's versions intersect. If the symbolic values of the existing invariant candidate and input invariant candidate are not the same, Algorithm 13 is called at line 13 to remove any conflicting invariant candidate. In case there is no conflict, the preexisting invariant candidate's `Version` attribute is updated to be the union of the preexisting and input invariant candidates' `Version` attribute. After the loop, at line 21, if `invExist` is false, then the input invariant candidate is created at line 22 and added to the list of invariant candidates at line 23.

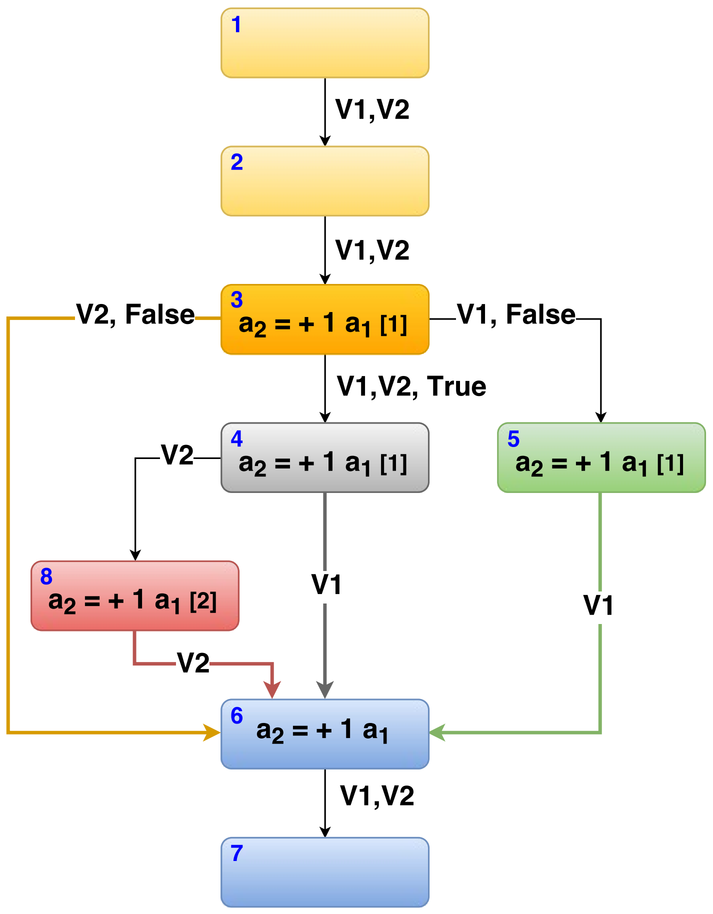


Figure 4.11: Query state; Snapshot 2

In our example, when the orange query reaches node 1, there were no preexisting invariant candidates. Hence, we add the resolution returned by the query as an invariant candidate. The state of the queries after this update is shown in Figure 4.12. The yellow cloud `[a = 1]`, represents the invariant candidate for the first version (as indicated in the brackets). Wherever the symbolic value represents more queries than the query being represented by the nodes color, the query's color has been appended (in abbreviation) to the symbolic value.

When the work-list is exhausted, in our example, the state of the query is shown in Figure 4.13. Since no conflicting queries were encountered during the propagation,

Algorithm 11 Algorithm to update the Invariant Candidates

```
1: function UPDATEINVARIANTS(InputCandidate, Versions)
2:     if Candidates is empty then
3:         InvCan $\leftarrow$ CREATEINVARIANT(InputCandidate, Versions)
4:         Add InvCan to Candidates
5:     else
6:         invExist $\leftarrow$ False
7:         for all InvCan $\in$ Candidates do
8:             if InvCan.Variable = InputCandidate.Variable then
9:                 if InvCan.Template = InputCandidate.Template then
10:                    if InvCan.Version $\cap$ Versions $\neq \emptyset$ then
11:                        invExist $\leftarrow$ True
12:                        if InvCan.InvariantCandidate $\neq$ InputCandidate then
13:                            REMOVEINVARIANT(InputCandidate.Variable,Versions)
14:                        else
15:                            InvCan.Version $\leftarrow$ InvCan.Version$\cup$InputCandidate.Version
16:                        end if
17:                    end if
18:                end if
19:            end if
20:        end for
21:        if invExist $\neq$ True then
22:            InvCan $\leftarrow$ CREATEINV(InputCandidate,Versions)
23:            Add InvCan to Candidates
24:        end if
25:    end if
26: end function
```

the invariant candidate is confirmed to be an invariant for both the versions and represented by the green colored cloud. At line 16, in Algorithm 6, the Algorithm 12 is used to output the result. As the invariant was established over the two versions we were interested in, it reports that there is no change of invariants for the variable ‘`a`’ between the two versions.

The Algorithm 12 outputs the change of invariants at the specified matched program point. If the invariant hasn't changed over the versions we evaluated, then invariant's `Version` attribute must be same as what we are evaluating over. Otherwise, there is a change in the invariants across the program versions. If there is a change of invariant then line 4 and 5 reports the change along with necessary information to

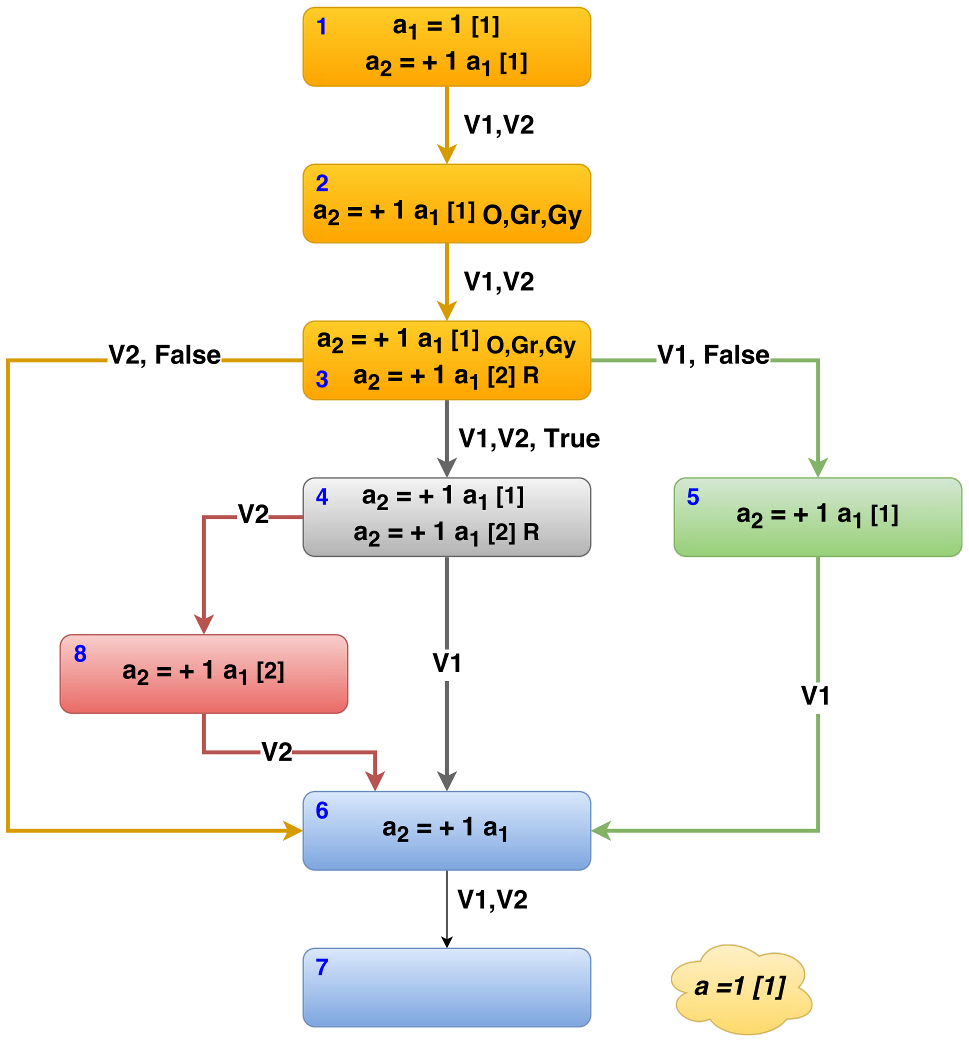

1 $a_1 = 1$ [1] $a_2 = +1\ a_1$ [1]
V1,V2
2 $a_2 = +1\ a_1$ [1] O,Gr,Gy
V1,V2
3 $a_2 = +1\ a_1$ [1] O,Gr,Gy $a_2 = +1\ a_1$ [2] R
V2, False
V1, False
V1,V2, True
4 $a_2 = +1\ a_1$ [1] $a_2 = +1\ a_1$ [2] R
5 $a_2 = +1\ a_1$ [1]
V2
8 $a_2 = +1\ a_1$ [2]
V1
V1
V2
6 $a_2 = +1\ a_1$
V1,V2
7
a =1 [1]

Figure 4.12: Query state; Snapshot 3

debug, understand or specify the change. Depending on the strength of the constraint solver and type of invariant template used, it is possible that certain type of invariants are never resolved. In this case, no change of invariant will be reported.

Now that we have a clear idea of how the algorithm works when there is no change of invariant, let's consider a case where there is one. The color coding and annotation methodology used in the previous figures will be re-used to save some textual explanations.

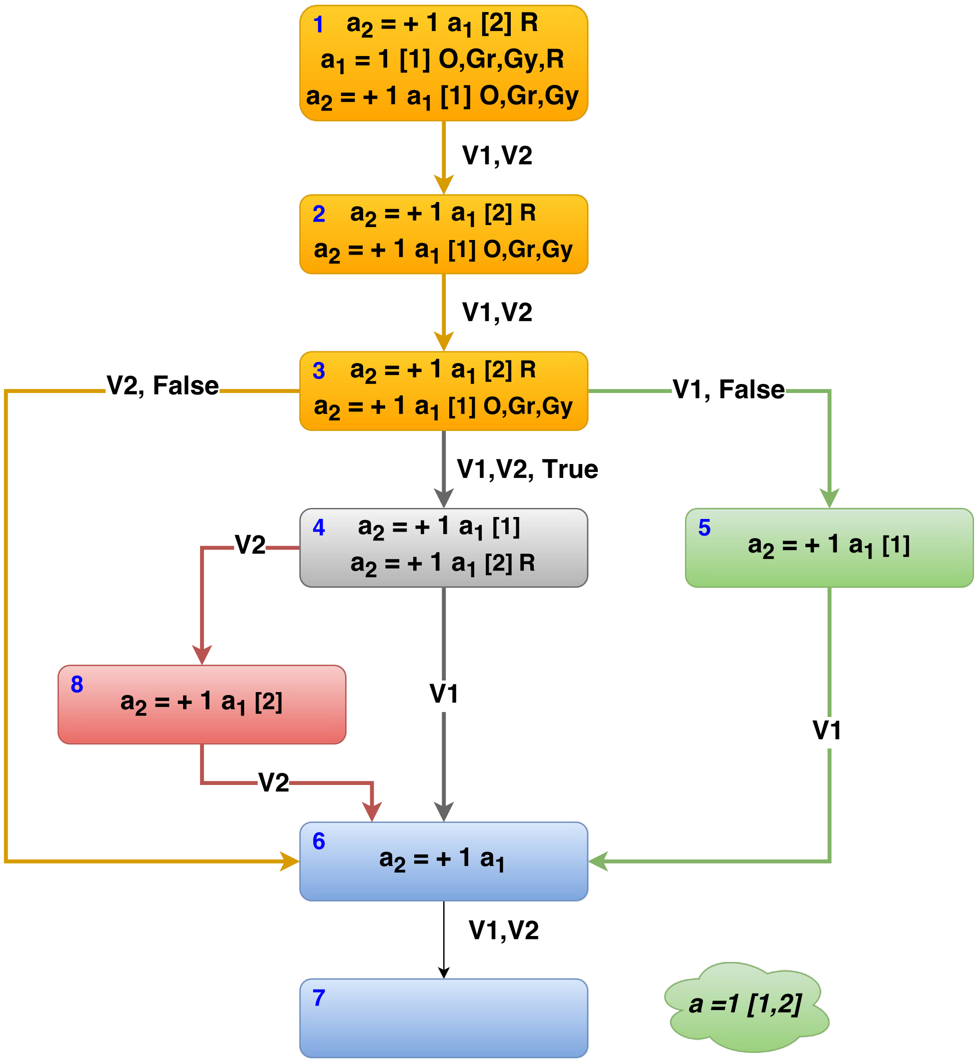


Figure 4.13: Query state; Snapshot 4

Operating under the same assumptions and constraints, let's analyze the change of invariants associated with the variable 'b' in the MVICFG. We will pickup the propagation from the state shown in Figure 4.14.

When the query from node 4 (gray) propagates to node 3, it will update the symbolic value of the query variable 'b' at node 3 to be [b_2 = + 1 b_1] for the version 1. Similarly, when the query from node 5 (green) propagates to node 3, it will try to update the symbolic value of query variable, again, as [b_2 = - 1 b_1] for the same version. Since the two symbolic values are conflicting, both the queries will be im-

Algorithm 12 Algorithm to output change of Invariants

```
1: function OutputChangeOfInvariant(V)
2:     for all InvCan $\in$ Candidates do
3:         if InvariantCandidate.Versions $\neq$ V then
4:             *Report Invariant as changed*
5:             *Reports the Variable, Symbolic Value and Version(s) for which the value holds*
6:         end if
7:     end for
8: end function
```

mediately canceled by Algorithm 9 and Algorithm 13 will be called to remove any conflicting invariant candidates.

Algorithm 13 Algorithm to remove an Invariant Candidates in case of conflict

```
1: function RemoveInvariant(Variable, Versions)
2:     for all InvCan $\in$ Candidates do
3:         if InvCan.Variable = InvariantCandidate.Variable then
4:             if InvCan.Template = InvariantCandidate.Template then
5:                 *Remove intersecting version from* InvCan.Version
6:             end if
7:         end if
8:     end for
9: end function
```

Algorithm 13 is used to remove invariant candidates in case of conflict. If there exist an invariant candidate with the same variable and template as the input parameters, then at line 5, we remove the intersecting versions from the preexisting invariant candidate's `Version` attribute. If after removing the intersecting versions, the preexisting invariant candidate is left with an empty `Version` attribute, then we remove the entire invariant candidate from the list of invariant candidates.

In this example, however, no candidates had been established and hence nothing needs to be removed. When all the queries have been exhausted the state of the algorithm will be as shown in Figure 4.15. The conflicting symbolic values at node 3 have been highlighted in red. No query got propagated to node 1 as the query was resolved at node 2 itself. Also, notice how the green and gray queries were not

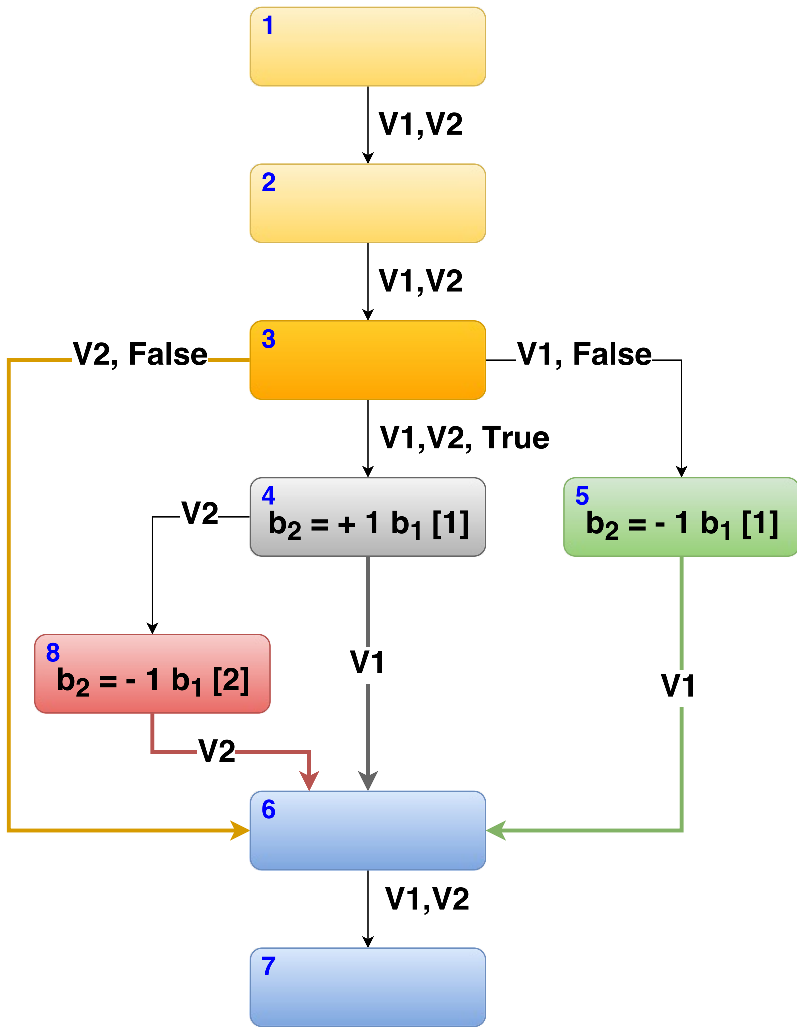


Figure 4.14: Query state; Snapshot 5

propagated after node 3 due to their conflicting nature. The Algorithm 12, will report the change of invariant as `[No Invariant; Added; b = 1]`.

With this clear understanding of the algorithms, let's proceed to their evaluation in the next chapter.

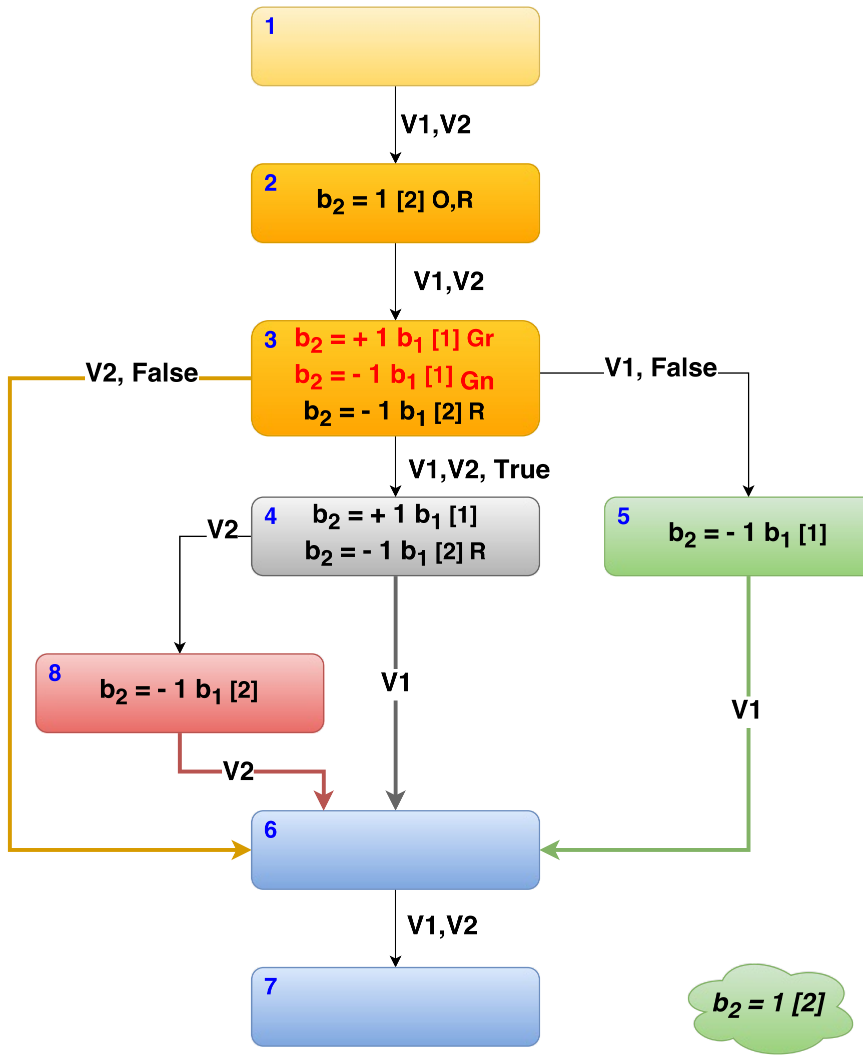


Figure 4.15: Query state; Snapshot 6

CHAPTER 5. EXPERIMENTAL RESULTS

The general goals of our experiments are to demonstrate that Hydrogen can compute the change of invariants correctly and to show their usefulness in understanding the changes made between different version of a program.

5.1 Implementation And Experimental Setup

Hydrogen is composed of mainly there major subsystems; LLVM compiler, MVICFG system and Z3 theorem prover. Intuitively, the MVICFG system uses the LLVM generated ICFG to create a Multi Version Inter-procedural Control Flow Graph over the program versions. This MVICFG is then traversed to evaluate the invariant templates by the Z3 based constraint solver.

LLVM compiler is used to compile and convert the input program versions into LLVM IR format. Specifically, we make use of the human readable assembly language representation of LLVM IR, which makes it easier to visualize and debug different versions of a program. The type-safe SSA representation in LLVM IR makes tracking and updating symbolic values of variables, using constraint solver like Z3, straightforward. The LLVM front-end provides a static analysis engine that is capable of providing dependency analysis, call graphs, loop initialization conditions, dominance and post-dominance trees, control flow graphs and basic pointer aliases information. In Hydrogen, we use Clang, the `C/C++` front-end of LLVM, for generating the ICFG required by Algorithm 2. Since the compilation flags used by a program can vary between different versions of a program, we decoupled the compilation stage of input program versions from Hydrogen. In effect, Hydrogen takes as input the compiled LLVM IR of different program versions.

The MVICFG system in Hydrogen acts as an arbitrator between LLVM and Z3 to facilitate the computation of change of invariants. It is written entirely in `C++` to make use of the traversal and analysis libraries provided by LLVM. The algorithms in Chapter 4 operate under the assumption that every instruction in MVICFG is associated with a location. Even though LLVM is capable of embedding location information into the compiled LLVM IR for most instructions, it is not complete. The MVICFG system is designed to overcome this shortcoming by intelligently extracting the missing location information directly from the source code. In addition to generating the MVICFG of the input program versions, it also provides traversal and node matching mechanism required by the algorithms in Section 4.3. The MVICFG system is also capable of exporting the generated MVICFG in `DOT` representation for visualizing the evolution of the program.

Hydrogen uses Z3 to track and update the symbolic value of the variables of interest. As Z3 doesn't natively understand LLVM IR instructions, the IR instructions have to be first translated into a format compatible with Z3. Table 5.1 showcases some concrete examples to give an intuitive idea. Under *LLVM IR* a valid instruction is shown with its translated form shown under *Z3 Translation*. Since instruction of same type tend to have similar translations, they are grouped under *Instruction Type*. Any variable in the translation requires a data type to be specified. Z3 only supports two data types; integer and real. Additionally, the real data type in Z3 requires the input to be in string format when directly specifying the value. We see this in the translation for `fsub double`, `fmul float` and `store float`. The LLVM IR, on the other hand, depending on the value of the float/double variable, can represent the direct value in either an exponential or hexadecimal string format. For successful translation, both the data type (of the variables) and the parameters (of the instruction) have to be correctly extracted and encoded.

Table 5.1: LLVM IR to Z3 translation examples

Instruction Type	LLVM IR	Z3 Translation
Binary Operations	x = add nsw i32 y, z	x.int = y.int + z.int
	x = sub nsw i32 y, 1	x.int = y.int - 1
	x = fsub double y, 3.200000e+01	x.real = y.real - “32.0”
	x = fmul float y, 0x3FF3AE147AE147AE	x.real = y.real - “1.23”
	x = fdiv float y, z	x.real = y.real - z.real
Memory Access	store i32 3, i32* x	x.int = 3
	x = load i32, i32* y	x.int = y.int
	store float 1.000000e+00, float* x	x.real = “1.0”

We translate each instruction traversed by the static demand-driven algorithm in Hydrogen, individually. The demand-driven algorithm, to compute the change of invariant, traverses the MVICFG backward from the instruction of interest until the query is resolved. However, the constraint solver in Z3 is sensitive to the order in which the translated instructions are provided to it. Hence, if the instructions are directly fed to the constraint solver, due to the traversal direction used by the demand-driven algorithm, wrong inferences can be made. The Hydrogen framework make use of internal data structures to re-order the translations, in program order, before invoking the constraint solver. Once the instructions are in the correct order and in a compatible form, the Z3 based constraint solver can be used to check for the invariant templates.

As completely translating all the LLVM IR is beyond the scope of this thesis, only a handful of instructions have been translated. As the thesis focuses primarily on defining and computing the change of invariant, instructions handling numerical operations and branch conditions were focused. The decision was made primarily due to the fact that numerical constraint are easier to resolve. Table 5.2 shows all the translated LLVM IR grouped by their type. Currently, only one invariant template, $x = c$, has been implemented. In the template x is any variable and c is any constant. As with many static analysis tools, loops are unrolled twice and have not been modeled fully. In keeping with the goals of the experiment, the current implementation of Hydrogen is crafted to work with only two versions of a program. This is sufficient to demonstrate the algorithm's ability to compute the change of invariants and showcase the usefulness of the same in understanding the changes made between the two versions. However, the implementation can be expanded to work with more than two versions without much difficulty.

Table 5.2: List of LLVM IR supported; grouped by instruction type

Terminal Instruction	**Binary Operations**	**Memory Access and Addressing Operations**	**Other Operations**
`ret`	`add`	`alloca`	`icmp`
`br`	`fadd`	`load`	`fcmp`
`switch`	`sub`	`store`	`call`
	`fsub`		
	`mul`		

Continued on next page

Table 5.2 – continued from previous page

Terminal Instruction	**Binary Operations**	**Memory Access and Addressing Operations**	**Other Operations**
	`fmul` `udiv` `sdiv`		

The experiment is designed to show that the algorithm is able to compute the change of invariant and to indicate its usefulness in understanding the changes made between program versions. Due to the limitations mentioned above, evaluating the Hydrogen framework over real world programs is difficult. In order to overcome this, a small benchmark of 4 test programs were constructed to mimic the common human made error as mentioned by Fedora in **DefCon** 2012. In order to show the applicability in real world programs, $tcas$ from SIR benchmark was also included in this benchmark. Since Hydrogen is made on top of LLVM IR instructions, all the gathered results are in the IR format as well.

5.2 Experimental Results

In this section we present the experimental data gathered from the benchmark using Hydrogen.

5.2.1 Computing the change of invariants

In this experiment we wanted to evaluate the ability of the algorithm to compute the change of invariants between two versions of a program. Table 5.3 shows the data collected from Hydrogen when evaluating the benchmark. Since we are using a template based invariant query, we have to check for the change of invariants on all the live variables present at the Matched Point. The number under *P* denote the program number in the benchmark. Under *Change of Invariant* we present the 4 tuple that defines it, namely *Matched Point*, *Version 1*, *Type* and *Version 2*. We report the line number associated with matched node in *Matched Point*. The invariants present, at the matched program point, for the different versions are reported under *Version 1* and *Version 2*, respectively. *Type* represent the change in relation between the variables present at the matched points across versions.

Table 5.3: Computed change of invariant for benchmark

P	**Changed Inv**	**Live Vars**	**Change of Invariants**			
			Matched Point	**Version 1**	**Type**	**Version 2**
1	1	3	10	%6_main = Input	M	%6_main = %2_main
	2	10	11	%10_main = Input	M	%10_main = %2_main
				%6_main = Input	M	%6_main = %2_main
2	1	3	18	%13_main = 1	M	%13_main = Input
	1	17	24	%27_main = 1	R	
	6	40	32	%47_main = Input	R	
				%50_main = 2	R	
				%45_main = 1	R	
				%30_main = 1	R	

Continued on next page

Table 5.3 – continued from previous page

P	**Changed Inv**	**Live Vars**	**Change of Invariants**			
			Matched Point	**Version 1**	**Type**	**Version 2**
				%27_main = 1	R	
3	2	9	12	%6_main = Input	M	%6_main = NDef
				%13_main = (+ (- (/ 160.0 9.0)) (* (/ 5.0 9.0) %3_main))	M	%13_main = (+ (- 462569 721726 8096000.0) %3_2_main)
4	1	8	21	%10_main = 1	R	
T	1	5	135		A	%92_alt _sep_test = 5

The results show that the algorithm is capable of computing the change of invariants, automatically, between different program versions. Upon manual inspection of the results, it was found that the effective number of change of invariants, in most cases, was one. Since LLVM IR is SSA based, each time a variable is assigned a new value, it gets renamed. This indicates that the number of change of invariants, between incremental versions, is generally small. We will discuss the cause and possible applications of the change of invariants reported in Table 5.3 in the Section 5.3.

In Table 5.4, we present the performance data gathered during the computation of change of invariants. According to the study by Marinescu et al. (2014), the average number of executable lines in a patch from 6 open source projects is six. As the test program are smaller, we limited the size of diff to an average of three. Similar to

Table 5.3, the number under *P* denote the program number in the benchmark and under *Matched Point* we report the matched program point's line number from the MVICFG. *Diff Size* shows the size of the diff file generated by standard Unix **diff** utility. Total execution time of Hydrogen, in ms, is presented in *Times*. The last two columns report the number of nodes visited for the query to be resolved for the first and second version respectively. The number reported in these columns are maximal if the same query was propagated in different paths in MVICFG causing different number of nodes to be traversed.

Table 5.4: Performance data for the static demand-driven Algorithm

P	**Diff Size**	**Matched Point**	**Time(ms)**	**No of Nodes Visited**	
				Version 1	**Version 2**
1	2	10	20.085	2	9
		11	107.827	11	17
2	4	18	75.677	5	6
		24	1755.08	6	23
		32	8015.64	32	48
3	4	12	177.21	18	16
4	2	21	95.893	7	8
T	5	135	41.705	14	12

The results suggest that the number of nodes traversed for the query resolution is generally small. The execution time depends on both the number of nodes traversed and the number of liver variables analyzed for the matched program point.

5.3 Interpreting The Change Of Invariants

In the previous section, we saw that Hydrogen was able to capture the change of invariants in the benchmark. The results presented in Table 5.3 are not very intuitive, as they are reported in LLVM IR format. This section interprets the change of invariants and show how they capture the changes made between the versions. According to a study by Nguyen et al. (2013), bug fixes repeat as much as 60% in some open source programs. Repetition of the fixes can be avoided if an assertion is inserted the first time the bug was fixed. In this spirit, we also show how the change of invariant can be used to generate assertions. We have grouped the discussion by programs, in the rest of this section.

5.3.1 Program P1

P1 is a program to convert temperature from Celsius to Fahrenheit. The difference between the two versions is shown below.

```
10d9
<       cin >> celsius;
```

From the diff generated, we can see that in the second version the statement seeking input to the variable `celsius` was deleted. In the LLVM IR, the variable `celsius` was given the name `%6_main`. The change of invariants reflects this change made between the versions. Specifically, the invariant at program point in line 10, changed from `[%6_main = Input]` to `[%6_main = %2_main]`, where `%2_main` is the uninitialized value of the variable. Since LLVM IR is in SSA form, the second use of the variable `celsius` got the name `%10_main`. This got reported as the second changed invariant in **P1**.

If the developer knows that the program behaves correctly in the first version, then the invariants generated for the correct version can be inserted as an assertion. In this specific case, if [celsius = Input] is inserted as an assertion before the use of the variable celsius, we can avoid bugs related to uninitialized variables in the future.

5.3.2 Program P2

P2 is a general arithmetic operations tester program. The diff over the program version is as follows.

```
17c17
<       if (number1 = 1) {
---
>       if (number1 == 1) {
```

There was a mistake in the if condition in the first version, causing the variable to be assigned instead of being compared. The variable number1 was renamed as %13_main in the LLVM IR. As evident from the change of invariant reported for **P2**, this change is captured by Hydrogen at program point in line 18. In this case, if the if condition was specified correctly, the symbolic value of the variable number1 would have been equal to input and not 1. The other change of invariants reported can be ignored as they are all associated with the same source variable number1, except for [%47_main = Input]. This invariant was present in the first version due to incorrect if condition. For this program, the change of invariants can be used characterize and understand the changes made between program versions but cannot be used directly to generate assertion like in **P1**.

5.3.3 Program P3

P3 is a program to convert temperature from Fahrenheit to Celsius. The difference between the two versions is shown below.

```
11c11
<       celsius = (fahrenheit − 32.0) * 5.0 / 9.0;
---
>       celsius = fahrenheit − 32.0 * 5.0 / 9.0;
```

A very simplified version of this program was explained in Chapter 3. Here, the source variable `celsius` was mapped to LLVM IR variable `%13_main` at program point in line 12. This change is reflected in the change of invariant reported. Like in **P1**, the invariants generated for the correct version, `[celsius = (fahrenheit - 32.0) * (5.0 / 9.0)]`, can be used as an assertion to prevent similar logical bugs in the future.

5.3.4 Program P4

P4 mimics, minimally, the scenario of missing `break` statement inside a `switch` condition. The diff over the program versions is as follows.

```
13a14
>                       break;
```

The first version of the program was missing a `break` statement in one of the switch cases. The fall-through caused by this missing statement eliminated one of the control flow paths, which would have negated the reported invariant in the first version. While the change is shown by the change of invariants, the exact cause is harder to debug. The path followed by the query, raised by the static demand-driven algorithm, is useful in debugging as it shows the paths along which computation of differing invariants went.

5.3.5 Program T

T is the `tcas` program from SIR benchmark. The difference between the two versions is shown below.

```
132c132,133
<               alt_sep++;
---
>               alt_sep += 1;
>               alt_sep -= 1;
```

The diff lines shown above occurred inside of a conditional branch. Since, in the second version, the value of the variable `alt_sep` was left unmodified inside the conditional branch, the invariant `[%92_alt_sep_test = 5]` was reported. In the first version, as the value was modified, it conflicted with the symbolic value from the other conditional path and negated the invariant. However, change of invariants reported in this program is not complete. There were other semantic changes that occurred between the versions which could not be captured due to the lack of mappings for arrays and `#define` variables.

In this section, we have seen that the change of invariants can be useful in understanding the changes made between different versions of a program and in some cases be used directly to generate assertions, required prevent similar bugs in the future.

CHAPTER 6. FUTURE WORK AND CONCLUSION

In this chapter, we discuss the future avenues for research using change of invariants and present a concluding message.

6.1 Future Work

The concept of change of invariants provides another tool to understand and analyze program evolution. New applications for the concept needs to be explored to demonstrate its usefulness in the long run. Since change of invariants fully utilizes the evolutionary aspect of the software, we hope that it will pave way for more longitudinal program analysis techniques.

The Hydrogen framework could be improved to support more invariant templates and Z3 translation. This will help to truly evaluate its potential in real world programs. The reporting and computation of the invariants have to be elevated from LLVM IR level to source level for the reported change of invariants to be useful for the developers.

Automatic assertion generation, using change of invariants, is an area which needs to be explored. Studies by Nguyen et al. (2010) and Kim et al. (2006) report that as much as 19-40% bug re-occur. Hence, providing a mechanism to automatically generate and update assertion with minimal interaction from the developers can help improve the quality of software evolution. As the number of change of invariants reported during the evaluation remained fairly low, it's use in automatically generating change contracts and differential assertions to aid in software verification needs to be explored further. For specifying change contracts and differential assertions, both longitudinal and latitudinal program property knowledge is required. The concept of change of invariant seems like an ideal fit in helping to automate these.

Another direction that we are interested in pursuing is computing the change of invariant for APIs. Already, works by Nguyen et al. (2014); Xie and Pei (2006) and Yang et al. (2006) try to mine the API invariants. But by computing interface level invariants, we can easily detect if incompatibility will arise from changing the versions of the API used. Also, by specifying the change of invariants at different program points, we might be better able to assist developer in adapting their program to accommodate changes to API or help the API developer classify a revision as major or minor release depending on if the revision breaks interface invariants or not.

6.2 Conclusion

Change of invariants provides a way of capturing the changes made between evolving versions of a program. A static demand-driven algorithm provides an efficient and scalable method to compute the change of invariants over program versions on top of MVICFG. The experimental results show that change of invariants can be calculated using the demand-driven algorithm and that they help in understanding the changes made between program versions.

The main contributions of this thesis are the definition for change of invariant, an efficient static demand-driven algorithm for computing them on top of multi version program representation called MVICFG and the prototype framework, called Hydrogen, that implemented algorithm to evaluate its ability to compute change of invariants.

BIBLIOGRAPHY

(2002). Two controlled experiments concerning the usefulness of assertions as a means for programming. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, ICSM '02, pages 84–, Washington, DC, USA. IEEE Computer Society.

Ammons, G., Bodík, R., and Larus, J. R. (2002). Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA. ACM.

Apiwattanapong, T., Orso, A., and Harrold, M. J. (2007). Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36.

Beyer, D., Henzinger, T. A., Majumdar, R., and Rybalchenko, A. (2007). Path invariants. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 300–309, New York, NY, USA. ACM.

Bozga, M., Habermehl, P., Iosif, R., Konečný, F., and Vojnar, T. (2009). *Automatic Verification of Integer Array Programs*, pages 157–172. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bradley, A. R., Manna, Z., and Sipma, H. B. (2006). *What's Decidable About Arrays?*, pages 427–442. Springer Berlin Heidelberg, Berlin, Heidelberg.

Buse, R. P. and Weimer, W. R. (2008). Automatic documentation inference for exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 273–282, New York, NY, USA. ACM.

Buse, R. P. and Weimer, W. R. (2010). Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA. ACM.

Casalnuovo, C., Devanbu, P., Oliveira, A., Filkov, V., and Ray, B. (2015). Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 755–766, Piscataway, NJ, USA. IEEE Press.

Colón, M., Sankaranarayanan, S., and Sipma, H. (2003). Linear invariant generation using non-linear constraint solving. In Hunt, WarrenA., J. and Somenzi, F., editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Berlin Heidelberg.

Cousot, P. and Cousot, R. (1977a). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA. ACM.

Cousot, P. and Cousot, R. (1977b). Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 77–94, New York, NY, USA. ACM.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA. ACM.

Csallner, C. and Smaragdakis, Y. (2006). Dynamically discovering likely interface invariants. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 861–864, New York, NY, USA. ACM.

Csallner, C., Tillmann, N., and Smaragdakis, Y. (2008). Dysy. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 281–290.

Daniel, B., Jagannath, V., Dig, D., and Marinov, D. (2009). Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 433–444, Washington, DC, USA. IEEE Computer Society.

Dillig, I., Dillig, T., Li, B., and McMillan, K. (2013). Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 443–456, New York, NY, USA. ACM.

Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B. (1980). Programming environments based on structured editors: The mentor experience.

Ernst, M. D. (2004). Invited talk static and dynamic analysis: Synergy and duality. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 35–35, New York, NY, USA. ACM.

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (1999). Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA. ACM.

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45.

Estler, H. C., Furia, C. A., Nordio, M., Piccioni, M., and Meyer, B. (2014). Contracts in practice. In *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442*, pages 230–246, New York, NY, USA. Springer-Verlag New York, Inc.

Flanagan, C. and Leino, K. R. M. (2001). Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK. Springer-Verlag.

Flanagan, C. and Qadeer, S. (2002). Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 191–202, New York, NY, USA. ACM.

Fluri, B., Wuersch, M., PInzger, M., and Gall, H. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743.

Fu, J., Bastani, F., and Yen, I.-L. (2008). Automated discovery of loop invariants for high-assurance programs synthesized using ai planning techniques. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 333–342.

Ghardallou, W. (2012). Using invariant relations in the termination analysis of while loops. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1519–1522.

Gupta, A. and Rybalchenko, A. (2009). Invgen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 634–640, Berlin, Heidelberg. Springer-Verlag.

Hangal, S. and Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM.

Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. (2012). Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58.

Henzinger, T. A., Hottelier, T., Kovács, L., and Voronkov, A. (2010). Invariant and type inference for matrices. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 163–179, Berlin, Heidelberg. Springer-Verlag.

Herzig, K., Greiler, M., Czerwonka, J., and Murphy, B. (2015). The art of testing less without sacrificing quality. In *Proceedings of the 2015 International Conference on Software Engineering*. IEEE – Institute of Electrical and Electronics Engineers.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Horwitz, S. (1990). Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 234–245, New York, NY, USA. ACM.

Horwitz, S., Prins, J., and Reps, T. (1989). Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387.

Jackson, D. and Ladd, D. A. (1994). Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM ’94, pages 243–252, Washington, DC, USA. IEEE Computer Society.

Kataoka, Y., Notkin, D., Ernst, M. D., and Griswold, W. G. (2001). Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM’01)*, ICSM ’01, pages 736–, Washington, DC, USA. IEEE Computer Society.

Kim, M. and Notkin, D. (2006). Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR ’06, pages 58–64, New York, NY, USA. ACM.

Kim, M., Notkin, D., and Grossman, D. (2007). Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE ’07, pages 333–343, Washington, DC, USA. IEEE Computer Society.

Kim, S., Pan, K., and Whitehead, Jr., E. E. J. (2006). Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT ’06/FSE-14, pages 35–45, New York, NY, USA. ACM.

Kovacs, L. I. and Jebelean, T. (2005). An algorithm for automated generation of invariants for loops with conditionals. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC ’05, pages 245–, Washington, DC, USA. IEEE Computer Society.

Kramer, J. and Cunningham, R. J. (1979). Invariants for specifications. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE ’79, pages 183–193, Piscataway, NJ, USA. IEEE Press.

Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. (2009). Loopfrog: A static analyzer for ansi-c programs. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 668–670.

Kusano, M., Chattopadhyay, A., and Wang, C. (2015). Dynamic generation of likely invariants for multithreaded programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 835–846.

Lahiri, S. K., Hawblitzel, C., Kawaguchi, M., and Rebêlo, H. (2012). Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg. Springer-Verlag.

Lahiri, S. K., McMillan, K. L., Sharma, R., and Hawblitzel, C. (2013). Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 345–355, New York, NY, USA. ACM.

Laviron, V. and Logozzo, F. (2009). Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation*, pages 229–244. Springer.

Le, T.-D. B., Yi, J., Lo, D., Thung, F., and Roychoudhury, A. (2014). Dynamic inference of change contracts. In *ICSME*, pages 451–455.

Le, W. and Pattison, S. D. (2014). Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1047–1058, New York, NY, USA. ACM.

Lin, C.-H., Liu, L., and Vasudevan, S. (2013). Generating concise assertions with complete coverage. In *Proceedings of the 23rd ACM International Conference on Great Lakes Symposium on VLSI*, GLSVLSI '13, pages 185–190, New York, NY, USA. ACM.

Loh, A. and Kim, M. (2010). Lsdiff: A program differencing tool to identify systematic structural differences. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 263–266, New York, NY, USA. ACM.

Marinescu, P., Hosek, P., and Cadar, C. (2014). Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 93–104, New York, NY, USA. ACM.

Massoni, T. (2007). An approach to invariant-based program refactoring. *Electronic Communications of the EASST*, 3.

Meyer, B. (1992). Applying'design by contract'. *Computer*, 25(10):40–51.

Miné, A. (2006). The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100.

Muşlu, K., Swart, L., Brun, Y., and Ernst, M. D. (2015). Development history granularity transformations. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, Lincoln, NE, USA.

Nagarajan, V., Gupta, R., Zhang, X., Madou, M., and De Sutter, B. (2007). Matching control flow of program versions. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 84–93.

Nguyen, H. A., Dyer, R., Nguyen, T. N., and Rajan, H. (2014). Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA. ACM.

Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., Nguyen, T., and Rajan, H. (2013). A study of repetitiveness of code changes in software evolution. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 180–190.

Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. (2012). Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 683–693, Piscataway, NJ, USA. IEEE Press.

Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N. (2010). Recurring bug fixes in object-oriented programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324, New York, NY, USA. ACM.

Nimmer, J. W. and Ernst, M. D. (2001). Static verification of dynamically detected program invariants: Integrating daikon and esc/java. *Electronic Notes in Theoretical Computer Science*, 55(2):255–276.

Nimmer, J. W. and Ernst, M. D. (2002). Invariant inference for static checking: An empirical evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20.

Notkin, D. (1985). The gandalf project. *Journal of Systems and Software*, 5(2):91–105.

Notkin, D. (2002). Longitudinal program analysis. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 1–1, New York, NY, USA. ACM.

Partush, N. and Yahav, E. (2014). Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 811–828, New York, NY, USA. ACM.

Păsăreanu, C. S. and Visser, W. (2004). Verification of java programs using symbolic execution and invariant generation. In *International SPIN Workshop on Model Checking of Software*, pages 164–181. Springer.

Perkins, J. H. and Ernst, M. D. (2004). Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 23–32, New York, NY, USA. ACM.

Person, S., Dwyer, M. B., Elbaum, S., and Păsăreanu, C. S. (2008). Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA. ACM.

Qi, D., Yi, J., and Roychoudhury, A. (2012). Software change contracts. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 22:1–22:4, New York, NY, USA. ACM.

Raghavan, S., Rohana, R., Leon, D., Podgurski, A., and Augustine, V. (2004). Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 188–197, Washington, DC, USA. IEEE Computer Society.

Ramanathan, M. K., Grama, A., and Jagannathan, S. (2007a). Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 240–250, Washington, DC, USA. IEEE Computer Society.

Ramanathan, M. K., Grama, A., and Jagannathan, S. (2007b). Static specification inference using predicate mining. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134, New York, NY, USA. ACM.

Reps, T. and Teitelbaum, T. (1984). The synthesizer generator. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 42–48, New York, NY, USA. ACM.

Rodríguez-Carbonell, E. and Kapur, D. (2005). Program verification using automatic generation of invariants. In *Proceedings of the First International Conference on Theoretical Aspects of Computing*, ICTAC'04, pages 325–340, Berlin, Heidelberg. Springer-Verlag.

Sagdeo, P., Athavale, V., Kowshik, S., and Vasudevan, S. (2011). Precis: Inferring invariants using program path guided clustering. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 532–535.

Sagdeo, P., Ewalt, N., Pal, D., and Vasudevan, S. (2013). Using automatically generated invariants for regression testing and bug localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 634–639.

Sankaranarayanan, S., Sipma, H. B., and Manna, Z. (2004). Non-linear loop invariant generation using gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 318–329, New York, NY, USA. ACM.

Schmitt, P. H. and Weiß, B. (2007). Inferring invariants by symbolic execution. *VERIFY*, 259:195–210.

Servant, F. and Jones, J. A. (2012). History slicing: Assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 43:1–43:11, New York, NY, USA. ACM.

Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., and Zheng, W. (2010). Do i use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 160–174, New York, NY, USA. ACM.

Shrestha, K. and Rutherford, M. J. (2011). An empirical evaluation of assertions as oracles. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 110–119, Washington, DC, USA. IEEE Computer Society.

Tao, Y., Dang, Y., Xie, T., Zhang, D., and Kim, S. (2012). How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11, New York, NY, USA. ACM.

Vasudevan, S., Sheridan, D., Patel, S., Tcheng, D., Tuohy, B., and Johnson, D. (2010). Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 626–629, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

Xie, T. and Pei, J. (2006). Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM.

Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. (2006). Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA. ACM.

Yang, W. (1991). Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755.

Yi, J., Qi, D., Tan, S. H., and Roychoudhury, A. (2013). Expressing and checking intended changes via software change contracts. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 1–11, New York, NY, USA. ACM.

Yi, J., Qi, D., Tan, S. H., and Roychoudhury, A. (2015). Software change contracts. *ACM Trans. Softw. Eng. Methodol.*, 24(3):18:1–18:43.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. (2011). How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA. ACM.

Zhang, Y. and Mesbah, A. (2015). Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 214–224, New York, NY, USA. ACM.