8-2015

Hydrogen: A Framework for Analyzing Software Revision Histories

Shannon D. Pattison

Hydrogen: A Framework for Analyzing Software Revision Histories

by

Shannon D. Pattison

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Software Engineering

Supervised by

Dr. Matthew Fluet

Department of Software Engineering

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

August 2015

The thesis “Hydrogen: A Framework for Analyzing Software Revision Histories” by Shannon D. Pattison has been examined and approved by the following Examination Committee:

Dr. Matthew Fluet
Assistant Professor
Thesis Committee Chair

Dr. Daniel Krutz
Lecturer

Dr. Stephanie Ludi
SE Graduate Program Director

Dedication

Special thanks to Dr. Matthew Fluet for recognizing that my work as a Graduate Research Assistant should be accepted as my thesis. Thanks to Dr. Pengcheng Shi for your invaluable insights as an experienced and exemplary researcher. Thank you Erika Mesh, Zachary Fitzsimmons, Naseef Mansoor, Ye Zhang, Sam Skalicky, Mohammed Alromaih, and Danilo Dominguez for your valuable feedback.

Acknowledgments

Dr. Wei Le contributed the idea for the Mulit-Version Interprocedural Control Flow Graph (MVICFG) as a graph with flow edges annotated with version numbers for a graph representation of multiple versions of a program. She also contributed the tool Marple used to perform demand driven analysis for patch verfication. Randall Goodman assisted with the implementation of the preprocessing scripts used for mining the source code repositories of benchmarks used in the results for this thesis and the related ICSE'14 technical paper. I contributed the formalization of the MVICFG, the algorithm to build the MVICFG, and the implementation of the Hydrogen framework.

Abstract

Hydrogen: A Framework for Analyzing Software Revision Histories

Shannon D. Pattison

Supervising Professor: Dr. Matthew Fluet

Hydrogen is a framework used for analyzing software revision histories for such applications as verifying bug fixes and identifying changes that cause bugs. The framework uses a graph representation of multiple versions of a program in a software revision history called a multi-version interprocedural control flow graph (MVICFG). The MVICFG integrates the control flow for multiple versions of a program into a single graph and provides a convenient way to represent semantic (i.e. control flow) change in a program. The MVICFG can also reduce the storage demands for representing the control flow for multiple versions of a program. Hydrogen implements an algorithm that uses data mined from source code repositories to construct the MVICFG. The MVICFG is analyzed using demand driven analysis for patch verification in multiple releases of software.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation for Hydrogen

Software aging is an observed phenomenon in software engineering where the quality of a software system gradually degrades with time [15]. There are generally two types of software aging. The first type is caused by the failure of software to keep up with changing needs. The second type of aging is caused by the changes to software in the software development lifecycle. Both can lead to rapid decline in software quality [16].

There is a need for tools that preserve or improve the quality of software to combat the effects of software aging. For example, code changes used to fix bugs are often incorrect and sometimes introduce new bugs. One study shows that 14.8%–24.4% of bug fixes in a sample of large open source projects are incorrect [20]. Another problem that contributes to software aging is the difficulty of understanding and assessing the impact of a change in software. A recent survey shows that developers find it difficult to determine whether a change will break code elsewhere [19]. There is also a need for better versioning systems for software [15].

The Hydrogen framework addresses problems in software aging by analyzing change in a key artifact in software revision histories – source code. The framework uses techniques in static analysis to interpret the semantics of source code change to detect bugs and validate changes that fix bugs.

The Hydrogen framework was created to analyze multiple versions of a program taken from a software revision history. A software revision history is a sequence of program versions where each successive version is created by making small incremental changes to an original version of a program. The Hydrogen framework is intended to represent a revision history at different levels of granularity. Hydrogen can be used to represent many versions of software where each successive version represents a small incremental change to an original version of a program. Hydrogen can also be used to analyze a few versions of a program where a version is a major or minor release of a program with a large amount of change between versions. A version in Hydrogen represents change in a program and each version is selected and assigned different meanings depending on the application of the Hydrogen framework.

Hydrogen uses an intermediate representation of a program called a *Multiple Version Interprocedural Control Flow Graph (MVICFG)*. The MVICFG is a graph with relations on the union of sequences of *Control Flow Graphs (CFG)* where a set of version numbers is related to an edge in the union of the edge sets from the CFG's. Intuitively, the MVICFG is a set of the unions of CFG's with each edge annotated to show what program versions the control edges and nodes belong. Each node in the MVICFG represents a statement common to at least one version of a program, and each edge in the MVICFG represents a control flow transition common to at least one program version.

An important application of the Hydrogen framework is patch verification for multiple software releases [14]. Specifically, the Hydrogen framework can be used to verify that a patch eliminates a bug in multiple software releases without introducing any new bugs. In other words, the framework is used to determine whether the patch is applicable to a buggy release and simultaneously verifies the patch for all buggy releases. A demand driven, path sensitive symbolic analysis is used on the MVICFG for detecting bugs in software changes.

The important applications of the Hydrogen framework can be summarized in the following list.

1. **Representing control flow change:** The MVICFG is a representation of control flow changes across multiple versions of a program that impact the behavior of a

program. This representation can be statically analyzed to assess the impact of program changes. The MVICFG can also be used to visualize path changes in the control flow of a program.

2. **Efficient and precise change verification:** Prediction of runtime behavior is done by applying interprocedural, path-sensitive, symbolic analysis to the MVICFG to precisely identify bugs and verify correctness of software changes. Analysis is done along changed program paths relevant to software revisions for greater efficiency. Efficiency is also improved by caching intermediate results when analyzing successive versions of a program in a revision history.

3. **Longitudinal analysis:** The MVICFG allows analysis of software change across multiple versions of a program to discover commonalities, differences, and the progression of changes in program properties (e.g., bugs and invariants) in a revision history.

4. **Facilitating online comparisons:** The MVICFG reduces the amount of control flow graph data involved in an analysis across multiple versions of a program by eliminating redundant nodes and edges common to multiple program versions. It is easier to determine commonalities between program versions by analyzing shared nodes and edges in the MVICFG that represent shared code across multiple versions. This eliminates a common problem in other approaches that repeatedly analyze shared code.

The main contribution of this thesis is a precise definition for the MVICFG and an efficient algorithm for constructing the MVICFG from source code repositories. An application of the MVICFG is demostrated using the Hydrogen framework to verify the correctness of changes for multiple versions of a program. Thus, the MVICFG is demonstrated as an effective data structure that can be used with new techniques in path sensitive analysis that are used to verify the correctness of changes and help maintain a baseline of quality.

1.2 Research Questions

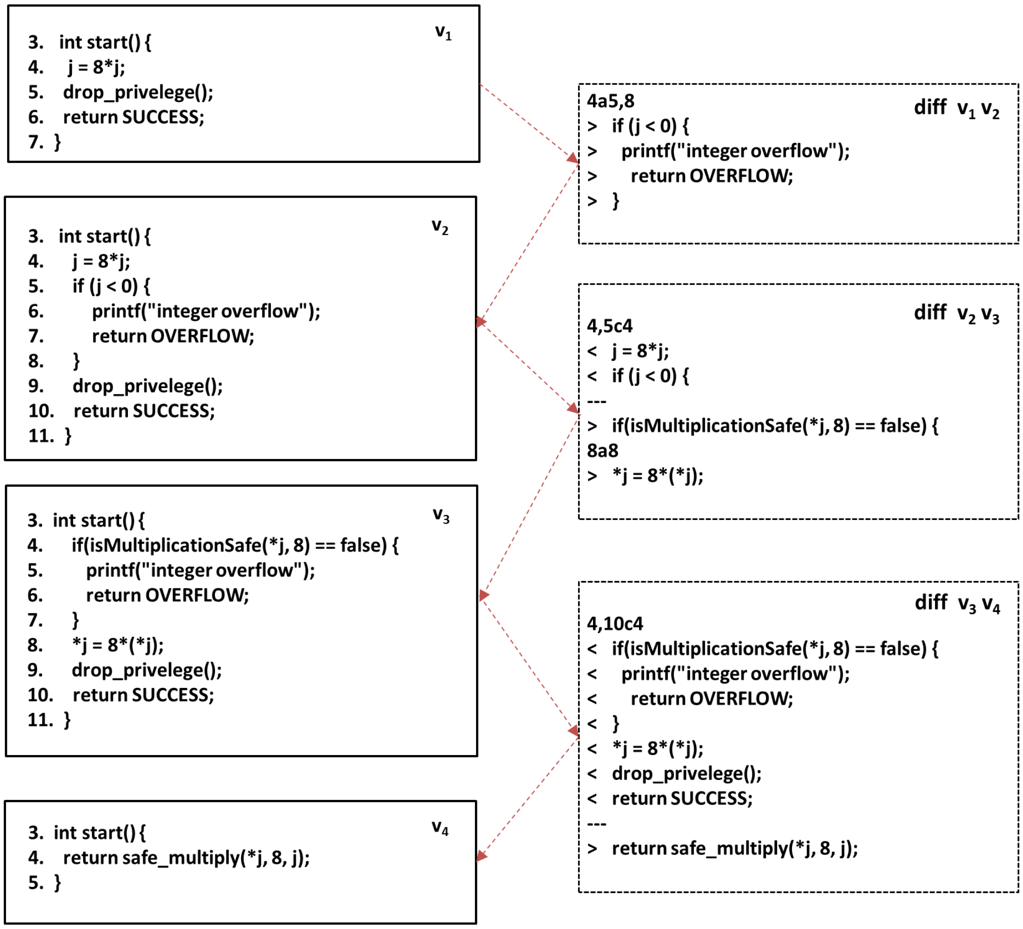
The ultimate goal of the Hydrogen framework is to reduce the effects of software aging and provide a better approach for maintaining or improving software quality in the software development lifecycle. The research questions in this study are concerned with the potential for using the MVICFG to support more efficient methods for achieving that goal. The research questions addressed in this report are the following.

1. What are some general methods that can be used to efficiently translate program revisions to control flow changes for a large number of incremental program revisions?
2. What are some well defined representations of sequences of interprocedural control flow graphs which would reduce storage or memory demands?
3. What data structures would improve the efficiency of path sensitive analysis across multiple versions of a program?
4. What methods can be used for mining source code repositories to represent control flow change over a large number of revisions in a software history?

1.3 Overview of the Hydrogen Framework

The MVICFG is used to represent both interprocedural and intraprocedural changes of a program across multiple versions. The purpose of Hydrogen is to apply demand driven analysis to the MVICFG for such applications as verifying bug fixes for bugs detected with an interprocedural path sensitive analysis. Thus, it is not only important to understand how the MVICFG represents intraprocedural changes, but it is also important to understand how changes to the procedures of a program (i.e. added and deleted procedures) are represented.[1]

[1] An overview of the MVICFG and its application to patch verification is available in section 2 of the publication related to the study of this thesis [14]. The overview in the publication focuses on how the MVICFG represents change within a singe procedure of a program. Section 2 in the publication also shows



```
3.  int start() {
4.    j = 8*j;
5.   drop_privelege();
6.   return SUCCESS;
7.  }
```

```
4a5,8
>  if (j < 0) {
>    printf("integer overflow");
>      return OVERFLOW;
>  }
```

```
3.   int start() {
4.     j = 8*j;
5.     if (j < 0) {
6.        printf("integer overflow");
7.        return OVERFLOW;
8.     }
9.     drop_privelege();
10.    return SUCCESS;
11.  }
```

```
4,5c4
<  j = 8*j;
<  if (j < 0) {
---
>  if(isMultiplicationSafe(*j, 8) == false) {
8a8
>  *j = 8*(*j);
```

```
3.  int start() {
4.     if(isMultiplicationSafe(*j, 8) == false) {
5.        printf("integer overflow");
6.        return OVERFLOW;
7.     }
8.     *j = 8*(*j);
9.     drop_privelege();
10.    return SUCCESS;
11.  }
```

```
4,10c4
<  if(isMultiplicationSafe(*j, 8) == false) {
<    printf("integer overflow");
<      return OVERFLOW;
<  }
<  *j = 8*(*j);
<  drop_privelege();
<  return SUCCESS;
---
>  return safe_multiply(*j, 8, j);
```

```
3.  int start() {
4.    return safe_multiply(*j, 8, j);
5.  }
```

Figure 1.1: Four versions of a simple program with correct and incorrect patches for an integer overflow.

There is one point of possible confusion to be aware of before reading on. The abbreviations MVCFG and MVICFG are used to discuss two types of graphs. A Multi-Version Control Flow Graph (MVCFG) refers to the control flow graph with annotated edges used to represent change within a single procedure. The MVICFG is easiest to understand as a

a simple example of how demand-driven, path sensitive symbolic analysis can be used on an MVICFG to verify bug fixes. A different overview is provided in this report that will describe how the MVICFG is used to represent both interprocedural and intraprocedural changes of a program across multiple versions.

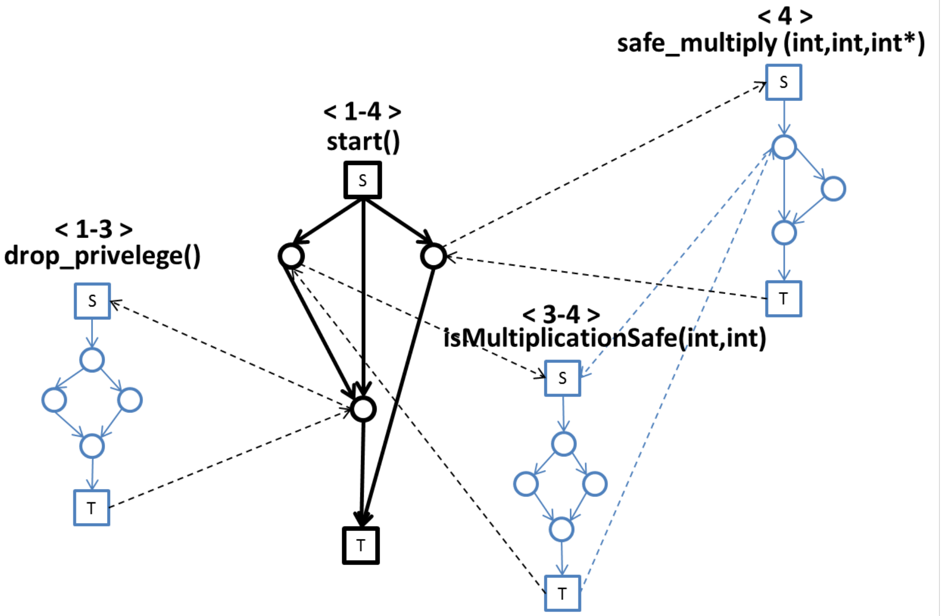


Figure 1.2: A high level view of the MVICFG for the four versions of the simple program in Figure 1.1.

graph composed by connecting multiple MVCFGs. [2] This distinction will become important in the next chapter of this thesis where the MVICFG is formally defined.

The code sample in Figure 1.1 shows a simple program with both intraprocedural and interprocedural changes that can be represented by a multi-version interprocedural graph (i.e. MVICFG). In this example, a simple program performs integer multiplication that is susceptible to an integer overflow. The blocks of code in the left column of Figure 1.1 show four different versions of the *start()* procedure in a simple program. The right column shows the diffs that represent the changes between the successive versions of the program. The change in v_2 incorrectly patches the integer overflow, v_3 has a correct patch, and v_4 is a refactored version of the program.

The diffs in Figure 1.1 also show how procedure calls change between versions. The

[2] The publication with the results of this study does not make a distinction between the Multi-Version Interprocedural Control Flow Graph (MVICFG) and the Multi-Version Control Flow Graph (MVCFG) [14].

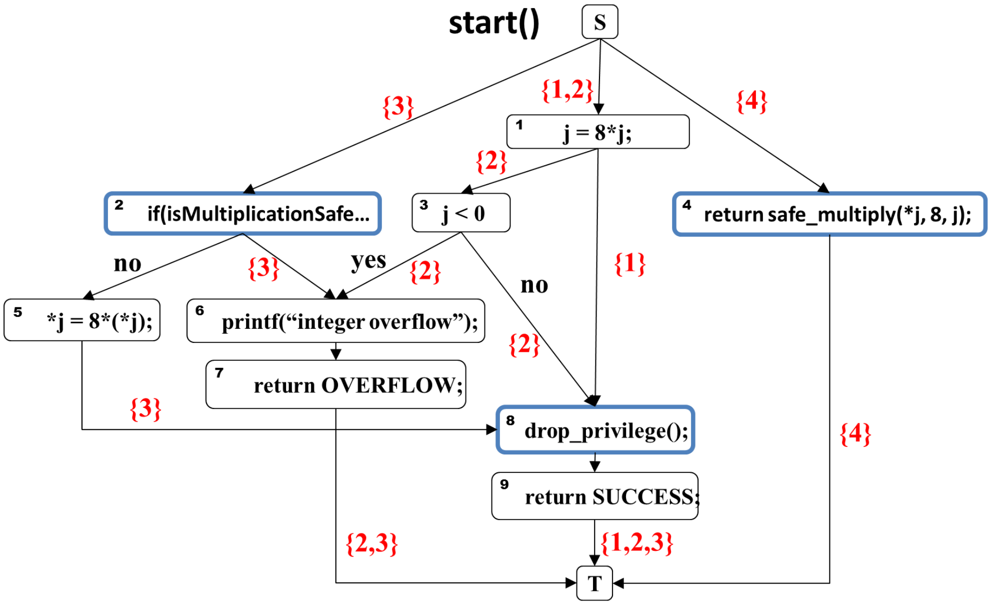


Figure 1.3: The MVCFG that represents all the changes within the $start()$ procedure.

start() procedure is the top level procedure of the program which calls the other procedures *drop_ privelege()*, *isMultiplicationSafe(int,int)*, and *safe_multiply(int,int,int*)*. The *drop_privelege()* procedure is deleted in v_4. The *isMultiplicationSafe(int,int)* procedure is added in v_3 on line 4 as part of the correct patch to guard against an integer overflow. The *safe_multiply(int,int, int*)* procedure is added in v_4 to encapsulate lines 4-8 and line 9 from v_3. These procedural changes are reflected in the MVICFG for the program.

The simple program from Figure 1.1 is represented by a MVICFG described in Figure 1.2 and Figure 1.3. Figure 1.2 is a high level view of the MVICFG which shows procedural changes in the program, and Figure 1.3 shows the annotated MVCFG representing changes in the $start()$ procedure of the program. Both graphs present different views of the MVICFG to provide information about both interprocedural and intraprocedural changes in the program.

The high level view in Figure 1.2 omits some details of the MVICFG to show how changes in the call flow graph are represented as procedures are added and deleted between versions of a program. All four procedures are represented in the MVICFG as MVCFGs labeled with the signature of their respective procedure. MVCFGs are connected by call flow edges that connect call sites within a calling procedure with the start and end nodes of the MVCFG for the called procedure. Each MVCFG in Figure 1.2 is labeled with a range of program versions in which the procedure exists. Notice that none of the MVCFGs for deleted procedures are ever removed from the MVICFG and are retained for a subset of program versions. The control flow paths of deleted procedures in the MVICFG never intersect the interprocedural paths from the versions of the program where the procedures were removed.

The graph in Figure 1.3 presents the Multi-Version Control Flow Graph for the $start()$ procedure with greater detail than in Figure 1.2. The graph is a union of the control flow graphs for each version of the *start()* procedure with the edges annotated with version numbers. The version numbers indicate to which versions each control flow edge belong, and these annotated edges are used to determine to which versions a node belongs. Edges without annotations belong to all versions. The control flow graph for any version can be identified simply by selecting the edges with an annotation that contains the version number and all nodes that are incident with the selected edges. These annotations can also be used to identify control flow changes in the *start()* procedure. For example, the control flow graph for v_1 is reconstructed with the nodes with indices 1, 8, and 9 and any incident edges with an annotation containing the number 1. Also notice that nodes 1, 8, and 9 are common to version 2. The control flow graph for version 2 also contain nodes 3, 6, and 7 that were inserted in version 2 to fix an integer overflow unsuccessfully. In version 3, nodes 1 and 3 are replaced by node 2, and node 5 is inserted before node 8. Node 2 is the correct patch for the integer overflow.

The overview of this section is intended to present the basic idea of the Multi-Version Interprocedural Control Flow Graph. This section also shows a simple example of how the MVICFG can be used to represent a patch in a program that would be analyzed interprocedurally. The next chapter presents a formal definition of the MVICFG that will be used to define an algorithm for constructing the MVICFG.

Chapter 2

Definition of the MVICFG

2.1 Preliminaries of the MVICFG

The following sections present a formal definition of the MVICFG using graph theory to establish a foundation for defining the algorithms used to construct an MVICFG. The notations and definitions in the following sections are used extensively in the definitions and descriptions of the algorithms in chapter 3. The definitions are also used to analyze the storage efficiency of the MVICFG and the efficiency and scalability of the algorithm used to construct the MVICFG. [1]

2.1.1 Definitions and Denotations

A graph is simple if it has no loops (i.e. an edge incident to one node) or parallel edges.

A directed graph G is an ordered pair (N, E) where N is a set of nodes and E is a set of directed edges. The edge set E is a set of ordered pairs of nodes (u, v) where $u, v \in N$.

A graph union of two graphs G_1 and G_2 is a new graph $G = G_1 \cup G_2$ where G is an ordered pair (N, E) and $N = N_1 \cup N_2$ and $E = E_1 \cup E_2$.

Given a graph $G = (N, E)$, the number of nodes in G is denoted by $|N|$, and the number of edges is denoted by $|E|$.

[1] The definition of the MVICFG in section 3 of the publication does not include the level of detail of the definitions presented in this chapter. The definition used in the publication defines the basic elements of the MVICFG needed to describe how the MVICFG is used in patch verification [14].

Given two sets of nodes N_1 and N_2, the number of nodes in a union of sets is $|N_1 \cup N_2| = |N_1| + |N_2| - |N_1 \cap N_2|$.

A graph union of a sequence of graphs $< G_1, G_2, ..., G_v >$ where $v > 0$ is the number graphs in a sequence is defined as $\bigcup\limits_{i=1}^{v} G_i = G_1 \cup G_2 \cup ... \cup G_v$.

A powerset $\mathcal{P}(S)$ of a set S is the set of all subsets of S. Given $S = \{x, y, z\}$ we have $\mathcal{P}(S) = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$.

A procedural program $\mathcal{R}$ is a set of procedures where each procedure $(x, P) \in \mathcal{R}$ is a pair where P is a sequence of $I_i \in P$ instructions where l is the number of instructions in P and $P =< I_1, I_2, ..., I_l >$ where $l \geq i > 0$ and x is a unique symbol used to name P.

Each procedure P is represented by a control flow graph (CFG). The CFG is a directed graph represented as a 4-tuple $C = (N, E, S, T)$ where (1) N is a set of nodes, (2) E is a binary relation on N such that $E \subseteq N \times N$ representing the set of all directed edges (arcs) in C, and (3) S and T are start and terminal nodes $S, T \in N$.

Each node n in the set of nodes N corresponds to an unique instruction I_i from a procedure P. Nodes $n \in N$ are labeled in a control flow graph C with a label function nl. Given the node $n \in N$ corresponding to the instruction $I_i \in P$, the labelling function maps the node to the index of I_i such that $nl(n) = i$.

The union of two control flow graphs (CFGs) is defined as follows. Let P_2 be a modified version of the procedure P_1. Let $C_2 = (N_2, E_2, S_2, T_2)$ and $C_1 = (N_1, E_1, S_1, T_1)$ be the CFGs for P_2 and P_1 respectively. The union $C_1 \cup C_2$ is defined as a new flow graph $(N_1 \cup N_2, E_1 \cup E_2, S, T)$. Given that the start and terminal nodes S and T are are always considered equivalent and labeled the same for any flow graph, we have $S = S_1 = S_2$ and $T = T_1 = T_2$.

The longest common subsequence (LCS) of two sequences is the longest sequence common to both sequences. For example, given two sequences TGCC and GCTA, the longest common subsequence is GC.

2.1.2 Multi-Version Graph

A graph union has the effect of integrating the nodes and edges from a sequence of graphs into a single graphical representation. This makes the graph union suitable for integrating the control flow for a sequence of CFGs. However, there is no way of distinguishing a single graph in the sequence from the other graphs in a sequence using the resulting graph from the graph union. The information about the graphs in the sequence is lost in the graph union. See Figure 2.1 (a) for an example of a graph union.

The graph union is extended by mapping sets of integers to edges to identify which edge sets the edges are members of in the original sequence of graphs. The labelling scheme works as follows. A set containing the index of a graph in a sequence is mapped to each edge in an edge set for each graph. An example of the edge labelling scheme can be seen in Figure 2.1 (b).

We formally define the edge labelling scheme as follows. We construct the mapping of edges to the graph index sets by first defining an edge labelling function. An edge labelling function is defined for each edge set E_i such that $a \in E_i$ *iff* $\alpha_i(a) = \{i\}$. We further extend the α_i function to include the domain of $a \in \bigcup\limits_{i=1}^{v} E_i$ by definition 2.1 as follows.

$$\alpha_i(a) = \begin{cases} \{i\} & a \in E_i \\ \emptyset & a \notin E_i \end{cases} \tag{2.1}$$

Using definition 2.1, we can construct a μ function that maps a set of graph index numbers $\{i_1, i_2, ..., i_u\} \in \mathcal{P}(\{1, 2, .., v\})$ to all edges in the union $\bigcup\limits_{i=1}^{v} E_i$. Let $a \in \bigcup\limits_{i=1}^{v} E_i$ be an edge in the union of edge sets and v be the number of graphs in a sequence. We define $\mu_v(a) = \{i_1, i_2, ..., i_u\}$ such that $\mu_v(a) = \bigcup\limits_{i=1}^{v} \alpha_i(a)$.

Using the μ function from our definition, we now define the multi-version graph for a sequence of graphs $< G_1, G_2, ..., G_v >$ as the ordered tuple $(\bigcup\limits_{i=1}^{v} N_i, \bigcup\limits_{i=1}^{v} E_i, \mu_v)$. We denote the multi-version graph as $\mathcal{M}$ in the following definition.

$$\mathcal{M} = (\bigcup_{i=1}^{v} N_i, \bigcup_{i=1}^{v} E_i, \mu_v) \tag{2.2}$$

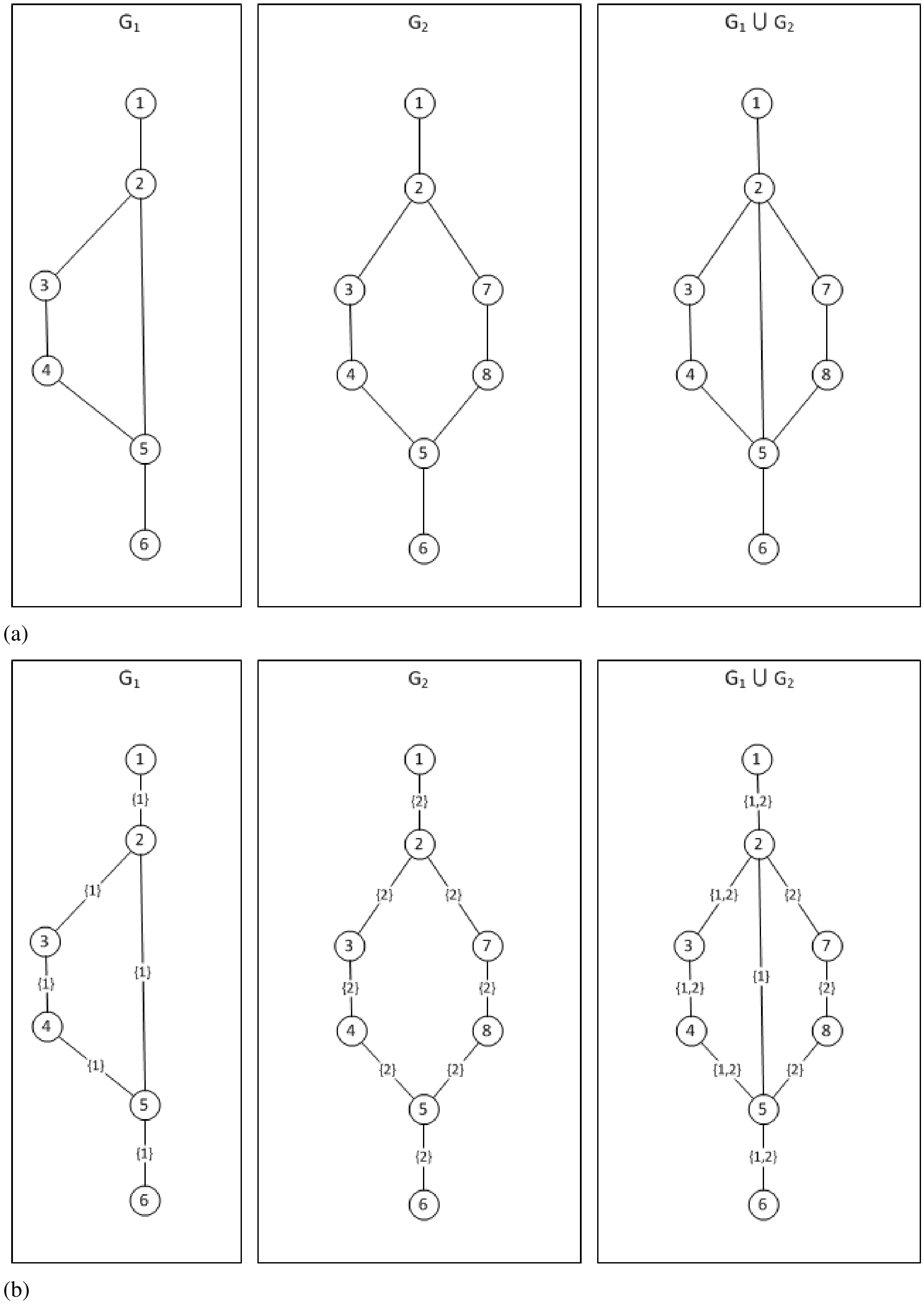


Figure 2.1: (a) shows the normal graph union $G_1 \cup G_2$, and (b) shows the graph resulting from the extended union $G_1 \cup G_2$ with edge labels.

Any graph $G_i = (N_i, E_i)$ can easily be reconstructed from the multi-version graph $\mathcal{M}$. The edge set E_i is reconstructed as follows. $\forall a \in \bigcup_{j=1}^{v} E_j$, if $i \in \mu_v(a)$, then $a \in E_i$. The node set N_i is reconstructed using incident edges from E_i. Given $\forall m, n \in \bigcup_{j=1}^{v} N_j$, if $\exists a \in E_i$ such that $a = (m, n)$ or $a = (n, m)$, then $m, n \in N_i$.

2.1.3 Graph Sequence Reduction Rule

The goal of the multi-version graph $\mathcal{M} = (\bigcup_{i=1}^{v} N_i, \bigcup_{i=1}^{v} E_i, \mu_v)$ is to represent $< G_1, G_2, ..., G_v >$ with a fewer number of elements than the number of elements in the sequence. While relationships $\sum_{i=1}^{v} |N_i| \geq |\bigcup_{i=1}^{v} N_i|$ and $\sum_{i=1}^{v} |E_i| \geq |\bigcup_{i=1}^{v} E_i|$ suggests a reduced number of elements in the multi-version graph, we cannot assume that there is a reduction for all sequences. We cannot ignore the contribution of $Domain(\mu_v)$ and $Range(\mu_v)$ of the relation μ_v to the total number of elements in $\mathcal{M}$ because representing μ_v takes non-trivial space. Thus, we have to count the number of elements in $Domain(\mu_v)$ and $Range(\mu_v)$ to determine when the multi-version graph reduces the number elements in a graph sequence.

Let $|Domain(\mu_v)|$ denote the number of elements in $Domain(\mu_v)$, and $|Range(\mu_v)|$ denote the number of elements in $Range(\mu_v)$. Also, let $\mathcal{M}$ be a multi-version graph. We define the number of elements in the multi-version graph as follows.

$$|\mathcal{M}| = |\bigcup_{i=1}^{v} N_i| + |\bigcup_{i=1}^{v} E_i| + |Domain(\mu_v)| + |Range(\mu_v)| \tag{2.3}$$

The number of elements $|Domain(\mu_v)|$ is straight forward. Since μ_v maps a set of graph index numbers $\{i_1, i_2, ..., i_u\}$ to all edges in the union $\bigcup_{i=1}^{v} E_i$, the number of elements $|Domain(\mu_v)|$ is the same as the number of edges in $\bigcup_{i=1}^{v} E_i$. Thus, $|Domain(\mu_v)| = |\bigcup_{i=1}^{v} E_i|$.

The number of elements $|Range(\mu_v)|$ is not so straight forward since $Range(\mu_v) \subseteq \mathcal{P}(\{1, 2, ..., v\})$. It's not enough to count the sets in $Range(\mu_v)$. It's important to count all the elements of the sets of graph indices in $Range(\mu_v)$ to count all elements in $\mathcal{M}$. Let $a \in \bigcup_{i=1}^{v} E_i$ be an edge in $\mathcal{M}$ and $\mu_v(a) = \{i_1, i_2, ..., i_u\}$ where $1 \leq u \leq v$ and

$\{i_1, i_2, ..., i_u\} \in \mathcal{P}(\{1, 2, ..., v\})$. Given $i_t \in \mu_v(a)$ where $1 \leq t \leq u$, we know that there is a one-to-one relationship between $a \in E_{i_t}$ and $i_t \in \mu_v(a)$. Thus, an $i_t \in \mu_v(a)$ is contributed to $Range(\mu_v)$ each time an edge a appears in an edge set E_i. Therefore, for all edges in the sequence of edge sets E_i, we have $|Range(\mu_v)| = |E_1| + |E_2| + ... + |E_v| = \sum\limits_{i=1}^{v} |E_i|$.

Given that $|Domain(\mu_v)| = |\bigcup\limits_{i=1}^{v} E_i|$ and $|Range(\mu_v)| = \sum\limits_{i=1}^{v} |E_i|$, we can reformulate definition 2.3 as follows.

$$|\mathcal{M}| = |\bigcup_{i=1}^{v} N_i| + \sum_{i=1}^{v} |E_i| + 2\,|\bigcup_{i=1}^{v} E_i| \tag{2.4}$$

Clearly the multi-version graph $\mathcal{M}$ is not a reduction for all sequences of graphs. Consider $< G_1, G_2, ..., G_v >$ to be a sequence of disjoint graphs. Then it holds that $\sum\limits_{i=1}^{v} |N_i| = |\bigcup\limits_{i=1}^{v} N_i|$ and $\sum\limits_{i=1}^{v} |E_i| = |\bigcup\limits_{i=1}^{v} E_i|$. If any G_i is not trivial, then $|Domain(\mu_v)| + |Range(\mu_v)| > 0$. Thus, given the number of elements in a sequence $N = \sum\limits_{i=1}^{v} |N_i| + \sum\limits_{i=1}^{v} |E_i|$, we have $N < |\mathcal{M}|$.

Given N is the number of elements in a sequence of graphs and $|\mathcal{M}|$ is the number of elements in the mult-version graph, we need $N > |\mathcal{M}|$ for there to be a reduction of the elements in a sequence of graphs. Given the previous definitions, we substitute for N and $|\mathcal{M}|$ to get $\sum\limits_{i=1}^{v} |N_i| + \sum\limits_{i=1}^{v} |E_i| > |\bigcup\limits_{i=1}^{v} N_i| + \sum\limits_{i=1}^{v} |E_i| + 2\,|\bigcup\limits_{i=1}^{v} E_i|$. The inequality simplifies to the following relationship.

$$\sum_{i=1}^{v} |N_i| - |\bigcup_{i=1}^{v} N_i| - 2\,|\bigcup_{i=1}^{v} E_i| > 0 \tag{2.5}$$

Assuming the graphs in the sequence are simple, we can use the first theorem of graph theory to reformulate the Principle 2.5 using the identity $2|\bigcup\limits_{i=1}^{v} E_i| = \sum\limits_{n \in \bigcup\limits_{i=1}^{v} N_i} deg(n)$. The $deg(n)$ represents the degree of a node from the graph of the union and not the degrees from the graphs of origin. Principle 2.5 is reformulated to the following.

$$\sum_{i=1}^{v} |N_i| - |\bigcup_{i=1}^{v} N_i| - \sum_{n \in \bigcup_{i=1}^{v} N_i} deg(n) > 0 \tag{2.6}$$

We call the principle in 2.6 the Graph Sequence Reduction Rule. Basically, this is the condition for the total number of elements in a sequence of graphs to be reduced by using the multi-version graph. Intuitively, the inequality in 2.6 holds when the intersection of all the node sets in the sequence is large and the graphs are sparse.

The principle in 2.6 can be extended to directed and non-simple graphs to make the principle more applicable to ICFGs. Assuming an ICFG can allow loops and parallel edges, we can use $\epsilon > 0$ to represent the number of non-simple edges in a graph. Given a sequence of directed graphs, the sum of degrees $\sum_{n \in \bigcup_{i=1}^{v} N_i} deg(n)$ for undirected graphs becomes $\sum_{n \in \bigcup_{i=1}^{v} N_i} id(n) + \sum_{n \in \bigcup_{i=1}^{v} N_i} od(n)$. Finally, we can express the Graph Sequence Reduction Rule for directed non-simple graphs with the following.

$$\sum_{i=1}^{v} |N_i| - |\bigcup_{i=1}^{v} N_i| - \sum_{n \in \bigcup_{i=1}^{v} N_i} id(n) - \sum_{n \in \bigcup_{i=1}^{v} N_i} od(n) + \epsilon > 0 \tag{2.7}$$

2.2 Multi-Version Control Flow Graph

The multi-version control flow graph is defined by extending the definition of the multi-version graph from the previous section to a sequence of CFGs. The trivial difference between directed graphs and CFGs is the designation of the start and terminal nodes S and T. However, the key problem with extending the definition of the multi-version graph is related to labeled nodes in CFGs.

Before we discuss the problem with labeled nodes in CFGs, we will clarify what we mean by an instruction. In our definition of the Multi-Version Control Flow Graph, we are not concerned with the operational semantics of any particular programming language.

An instruction in the context of the MVCFG is simply a string with an index for a position within a sequence of instructions for a procedure. The presumption is that a procedure in every procedural programming language is defined as a sequence of statements or expressions which translate into a sequence of instructions. Thus, a procedure P is defined as a sequence of $I_i \in P$ instructions where l is the number of instructions in P and $P =< I_1, I_2, ..., I_l >$ where $l \geq i > 0$. We are also not concerned with how a procedure P gets translated to a CFG. We always assume that for a given procedure P there is a corresponding flow graph C.

Recall from section 2.1.1 the definition of a control flow graph (CFG). The CFG is a directed graph represented as a 4-tuple $C = (N, E, S, T)$ where (1) N is a set of nodes, (2) E is a binary relation on N such that $E \subseteq N \times N$ representing the set of all directed edges (arcs) in C, and (3) S and T are start and terminal nodes $S, T \in N$. Each node n in the set of nodes N corresponds to an unique instruction I_i from a procedure P. Nodes $n \in N$ are labeled in a control flow graph C with a label function nl. Given the node $n \in N$ corresponding to the instruction $I_i \in P$, the labeling function maps the node to the index of I_i such that $nl(n) = i$. In other words, nodes in a CFG are labeled by the index of the instruction the node represents and not by the string representation of the instruction.

Assume we have two flow graphs C_1 and C_2. Although the union of the node sets $N_1 \cup N_2$ and edge sets $E_1 \cup E_2$ is based on the union of labeled graphs, the node labeling functions nl_1 and nl_2 are not consistent between flow graphs C_1 and C_2 respectively. This is because the instruction indices change between modifications from P_1 to P_2. For the nodes $m \in N_1$ and $n \in N_2$, it is often the case that $nl_1(m) \neq nl_2(n)$, but the nodes correspond to the same instruction $I_i \in P_1$ and $I_j \in P_2$ where $I_i = I_j$. Choosing the string representations of instructions as node labels instead of instruction indices does not solve the problem. The string representations of instructions cannot be used to uniquely label nodes because the same statement or expression for an instruction can be repeated in a procedural sequence. The location of an instruction is an identifying attribute. Thus, a new method for mapping nodes from N_1 to N_2 is required.

A mapping of nodes is created between flow graphs C_1 and C_2 using the longest common subsequence (LCS) of instructions from the procedures P_1 and P_2 using the string representation of each instruction. The LCS is used to identify which instructions remain unchanged between P_1 and P_2 and which instructions were inserted and deleted to modify P_1 into P_2. The nodes are mapped from N_1 to N_2 based on the LCS of string representations of instructions the nodes represent. A relation $\Phi : \mathbb{N} \rightarrow \mathbb{N}$ is created between instruction indices from P_1 to P_2 using the longest common subsequence. Given the Φ_1 mapping from node set N_1 to N_2 and the nodes $n_1 \in N_1$ and $n_2 \in N_2$, we have $n_1 = n_2$ *iff* $\Phi_1(nl_1(n_1)) = nl_2(n_2)$.

We use the node mapping Φ to define the union of the edge sets $E_1 \cup E_2$. Given $(m_1, n_1) \in E_1$ and $(m_2, n_2) \in E_2$, we have $(m_1, n_1) = (m_2, n_2)$ *iff* $\Phi_1(nl_1(m_1)) = nl_2(m_2)$ and $\Phi_1(nl_1(n_1)) = nl_2(n_2)$.

Now we can use an extended definition of the multi-version graph in (2.8) that integrates the control flow for all versions of a procedure. We simply extend the definition by adding the start and terminal nodes S and T and performing the graph union for the sequence of control flow graphs $< C_1, C_2, ..., C_v >$ using the node mapping functions $< \Phi_1, \Phi_2, ..., \Phi_{v-1} >$ in the following definition.

$$\mathcal{M} = (\bigcup_{i=1}^{v} N_i, \bigcup_{i=1}^{v} E_i, \mu_v, S, T) \tag{2.8}$$

Figure 2.2 shows an example of the union for two flow graphs C_1 and C_2. The flow graph C_1 represents the first version of the procedure P_1, and C_2 represents P_2 the modified version of P_1. The graphs C_1 and C_2 show the edges labeled with a set containing the graph version index. For C_1 and C_2, the graph shows each edge in the edge sets E_1 and E_2 are labeled with the sets $\{1\}$ and $\{2\}$ to show that $\alpha_i(a) = \{i\}$ for all $a \in E_i$. The arc labels in $C_1 \cup C_2$ show the labels for the edge set $E_1 \cup E_2$ where $a \in (E_1 \cup E_2)$ and $\mu_2(a) = \alpha_1(a) \cup \alpha_2(a)$. We first need to understand how nodes are mapped between node sets N_1 and N_2 of C_1 and C_2 to understand how the edge labels are created in $E_1 \cup E_2$.

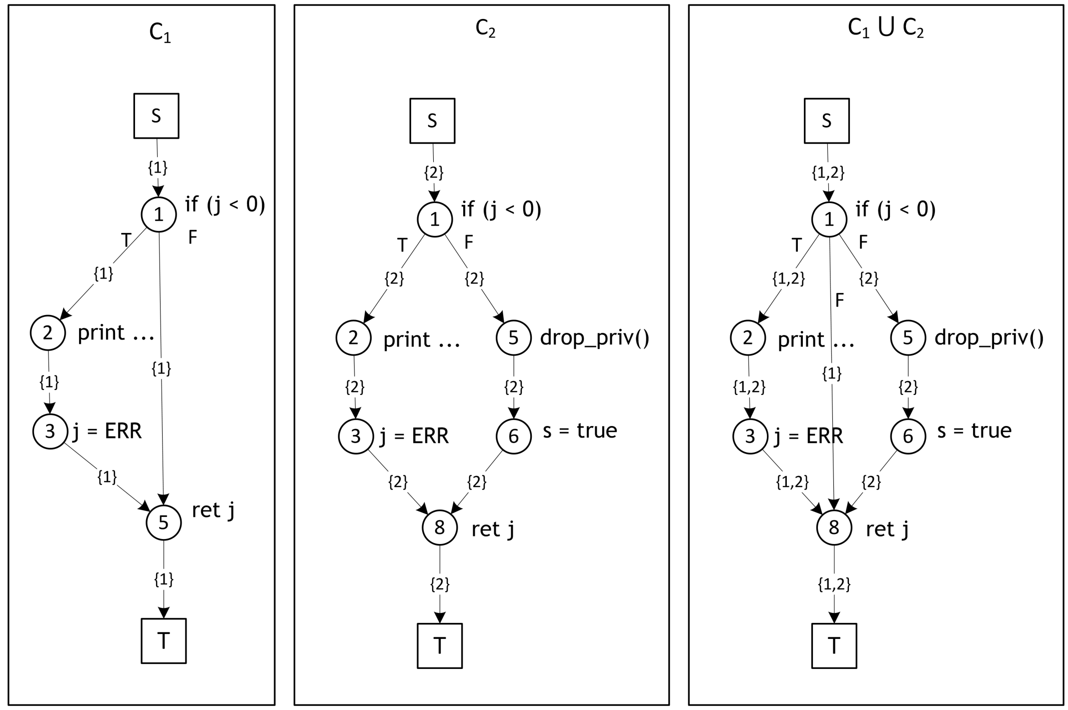
(a)

P_1

```
if $j < 0$ then
    print "overflow"
    $j = ERR$
end if
return $j$
```

P_2

```
if $j < 0$ then
    print "overflow"
    $j = ERR$
else
    drop_priv()
    $s =$ true
end if
return $j$
```



(b)

Figure 2.2: (a) shows the code for two versions of a procedure P_1 and P_2 represented by C_1 and C_2 respectively, and (b) shows the graph resulting from the extended union operation $C_1 \cup C_2$.

As mentioned previously, a node mapping between N_1 and N_2 is created by finding a longest common subsequence (LCS) of instructions for P_1 and P_2. Figure 2.2 shows that P_2 is created by inserting an else statement at location 4 in P_1. The LCS gives us the node mapping $\Phi_1 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 7, 5 \mapsto 8\}$. Using Φ_1, it is easy to determine which edges belong to $E_1 \cap E_2$ and which do not. The flow graph $C_1 \cup C_2$ in figure 2.2 shows how the edge label unions are done using the node mapping Φ_1. Figure 2.2 shows $\mu_2(a) = \{1, 2\}$ for all the edges $a \in (E_1 \cap E_2)$. For all the edges $a \in E_1$ and $a \notin (E_1 \cap E_2)$, we have $\mu_2(a) = \{1\}$. For all the edges $a \in E_2$ and $a \notin (E_1 \cap E_2)$, we have $\mu_2(a) = \{2\}$.

The multi-version control flow graph is the method for integrating the control flow for multiple revisions of a procedure in a software revision history. Although this is sufficient for creating a graphical representation of change at the intraprocedural level, it is not sufficient for describing change at the interprocedural level. Additional methods are needed to construct an MVICFG that graphically represents the change in call flow for procedures that are added or removed in software revisions. The next section defines the MVICFG, and the next chapter describes the algorithm used to construct an MVICFG that represents both interprocedural and intraprocedural changes.

2.3 Multi-Version Interprocedural Control Flow Graph

The definition of the Multi-Version Interprocedural Control Flow Graph (MVICFG) uses the union operation with Interprocedural Control Flow Graphs (ICFG) and builds on the definition of the Multi-Version Control Flow Graph (MVCFG) to represent both intraprocedural and interprocedural change. The MVICFG uses MVCFGs to represent all the procedures of a program for a sequence of versions and shows how the MVCFGs are interconnected in a call flow. The definition of the MVCFG is extended to include call flow information to represent both intraprocedural and interprocedural change in a procedure for multiple versions of a program.

Only a simple defiinition of an Interprocedural Control Flow Graph (ICFG) is needed to define a MVICFG. An ICFG is generally a graph that shows the control flow paths of a program which intersect multiple procedures. An ICFG not only shows all the control flow graphs for all the procedures in a program, it also shows how procedures are interconnected via the call flow of a program. The ICFG is essentially a set of all the control flow graphs for a procedural program with some additional information about the call flow for a program and a method for uniquely identifying control flow graphs by the procedure each graph represents. In order to define the ICFG as a set CFGs, the definition of the CFG from section 2.2 must be extended to include information about the call flow of a program.

The ICFG contains a unique CFG for every procedure in a program. The problem with uniquely identifying a control flow graph in an inteprocedural control flow graph is similar to the problem of uniquely identifying a procedure in a procedural program. Given a procedural program $\mathcal{R}$, it's possible to have two different procedures P' and P'' where both procedures have the same sequence of instructions $< I_1, I_2, ..., I_l >$. However, we do not define equality of procedures in the same way that sequences are normally considered equal. The procedures P' and P'' are identified by the unique symbols x' and x'' respectively. Typically, theses symbols represent the call signatures in a procedural program used to map a sequence of instructions for a procedure to a memory location in a program. For our purposes, we simply treat a symbol as an index number that is unique for each procedure in a program. In the context of a procedural program $\mathcal{R}$, the procedures P' and P'' are paired with their respective symbols such that $(x', P') \in \mathcal{R}$ and $(x'', P'') \in \mathcal{R}$. An unique symbol represents only one procedure in $\mathcal{R}$. Thus, when two procedures $(x', P') \in \mathcal{R}$ and $(x'', P'') \in \mathcal{R}$ are the same, we must satisfy the condition $x' = x'' \implies P' = P''$. The definition of a control flow graph needs to be extended to include symbols for the same reason two procedures in the same program require symbols. Since it is possible for two procedures to have the same control flow graphs, the symbol x for the procedure $(x, P') \in \mathcal{R}$ shall be included as an element in the definition of a CFG.

The definition of the CFG is also extended to include information about the call flow of a program in an ICFG. This is done by identifying all the nodes in the CFG that represent call instructions. Recall that for each instruction in a procedure $I_i \in P$, an instruction has two properties – a position i and a string representation. The call instruction has an additional property. The call instruction uses a symbol to identify the procedure that is called. The assumption is that the symbol of the called procedure is part of the string representation for the call instruction. The nodes representing call instructions are paired with the symbols of the callees from each call instruction. Given a control graph (N, E, S, T), the set of all call node pairs K include each call node $c \in N$ in a pair with a callee symbol x such that $(c, x) \in K$. The set of call node pairs K is included as part of the extended definition of the CFG.

An extended definition of a CFG is also needed to represent the call flow in an ICFG by including a call node mapping function. Given two procedures $(x, P'), (y, P'') \in \mathcal{R}$ where P' calls P'', let C' be the control flow graph for procedure P' and C'' be the control flow graph for procedure P''. Let K' be the set of pairs in C' where each pair $(c, y) \in K'$ contains the symbol y of the called procedure $(y, P'') \in \mathcal{R}$ and a node $c \in N'$ from the node set N' of C' representing the call instruction $I' \in P'$. The call node mapping function γ is used to represent the call flow edge in an ICFG where $\gamma[c] = C''$.

The definition of an ICFG that is used to define an MVICFG is as follows. An interprocedural control flow graph IC representing the procedural program $\mathcal{R}$ is defined as a set of control flow graphs $IC = \{C_1, C_2, ..., C_n\}$ where each control flow graph $C \in IC$ is represented as a 7-tuple $C = (N, E, S, T, K, x, \gamma)$ where (1) N is a set of nodes, (2) E is a binary relation on N such that $E \subseteq N \times N$ representing the set of all directed edges (arcs) in C, (3) S and T are start and terminal nodes $S, T \in N$, (4) K is a set of call node pairs where all call nodes $c \in N$ are paired with the symbol y representing the procedure being called such that $(c, y) \in K$, (5) x is a symbol used to reference the procedure $(x, P) \in \mathcal{R}$ that C represents, and (6) γ is a call node mapping function where $(c, y) \in K$ *iff* $\exists C' \in IC$ such that $\gamma[c] = C'$.

Symbols become important when constructing the MVCFG for multiple versions of control flow graphs in the construction of an MVICFG. Matching procedures across consecutive versions of a program uses the same principle of matching symbols to determine equality of two procedures within a single version of a program. Given two consecutive versions of a procedural program $\mathcal{R}_1$ and $\mathcal{R}_2$ and two procedures $(x_1, P_1) \in \mathcal{R}_1$ and $(x_2, P_2) \in \mathcal{R}_2$, we say that P_1 and P_2 represent two versions of the same procedure *iff* $x_1 = x_2$. The same applies for matching CFGs between consecutive versions of an ICFG across multiple versions of an ICFG in a sequence $< IC_1, IC_2, ..., IC_v >$. Given $C'_1 \in IC_h$ and $C'_2 \in IC_{h+1}$ where $C'_1 = (N'_1, E'_1, S, T, K'_1, x_1, \gamma'_1)$ and $C'_2 = (N'_2, E'_2, S, T, K'_2, x_2, \gamma'_2)$, C'_1 and C'_2 are two consecutive versions *iff* $x_1 = x_2$.

A mapping function δ is introduced to represent the mapping of CFGs between verisions of ICFGs using the symbol x from the definition of an ICFG. The δ function is needed to construct MVCFGs that represent a chain of CFGs in a subinterval of versions from a sequence of ICFGs. The δ function creates all the links between two subsequent ICFGs namely IC_1 and IC_2. Given a symbol x that exists in both IC_1 and IC_2, we have $\delta(x) = C$ where $C = (N, E, S, T, K, x, \gamma)$ and $C \in IC_2$.

We define δ as follows. Given the sequence of ICFGs where $\mathcal{I} =< IC_1, IC_2, ..., IC_v >$ and $1 \leq h < v$, let S_h be the set of all symbols in $IC_h \in \mathcal{I}$. Then δ_h is the function $\delta_h : S_h \to IC_{h+1}$ such that $(\forall x \in S_h)$:

$$\delta_h(x) = \begin{cases} (N, E, S, T, K, x, \gamma) & : \exists C \in IC_{h+1} \;\; s.t. \; C = (N, E, S, T, K, x, \gamma) \\ \emptyset & : \nexists C \in IC_{h+1} \;\; s.t. \; C = (N, E, S, T, K, x, \gamma) \end{cases} \tag{2.9}$$

The δ_h function is used to create the sequence of CFGs represented by each MVCFG contained in a MVICFG. The sequences of CFGs linked together using the sequence of relations $< \delta_1, \delta_2, ..., \delta_{v-1} >$ are called δ-chains, and a MVICFG contains a MVCFG for every δ-chain from the sequence $\mathcal{I}$. Every δ-chain begins with an initial $C_1 = (N, E, S, T, K, x, \gamma)$ where either $C_1 \in IC_h$ when $1 < h < v$ and $\delta_{h-1}(x) = \emptyset$ or $h = 1$. The δ-chains do not contain $\emptyset$. Thus, given the set of all symbols $\mathcal{S}$ for the sequence $\mathcal{I}$, if a symbol $x \in \mathcal{S}$ has more than one initial CFG where $C'_1 \in IC_h$ and $C''_1 \in IC_k$ such that $1 \leq h+1 < k \leq v$, then the CFGs belong to different δ-chains. Intuitively, this means that if a procedure referenced by symbol x is deleted, and a new procedure with the same symbol x is added in some subsequent version, then they will be referenced as two different procedures in the MVICFG.

Given a set of all symbols S for the sequence $\mathcal{I} =< IC_1, IC_2, ..., IC_v >$, the δ-chains for $x \in S$ are all the sequences on the subintervals of versions $[h, k]$ where $((h \leq k) \wedge (1 < h \wedge \delta_{h-1}(x) = \emptyset \vee h = 1) \wedge (k < v \wedge \delta_{k+1}(x) = \emptyset \vee k = v)$. Thus, for the subinterval $[h, k]$ and some initial $C_1 \in IC_h$ the δ-chain is $D =< C_1, \delta_h(x), \delta_{h+1}(x), ..., \delta_{k-1}(x) >$. The set $\mathcal{D}(\mathcal{I})$ is the set of all possible δ-chains for a sequence $\mathcal{I} =< IC_1, IC_2, ..., IC_v >$.

Using the extended definition of the control flow graph as a member of an ICFG, we can extend the definition of the MVCFG using the following definition. Given a sequence of interprocedural control flow graphs $\mathcal{I} =< IC_1, IC_2, ..., IC_v >$ and where v is the number of versions of a program being represented and the set of all δ-chains $\mathcal{D}(\mathcal{I})$, we represent a sequence of control flow graphs for a subinterval of versions as follows. Given $D \in \mathcal{D}(\mathcal{I})$ on a subinterval $[h, k]$ where $1 \leq h \leq k \leq v$, let $\mathcal{M}$ be the MVCFG that represents $D =< C_1, ..., C_j >$ where $j = k - h + 1$. We modify the definition 2.8 as follows.

$$\mathcal{M} = (\bigcup_{i=1}^{j} N_i, \bigcup_{i=1}^{j} E_i, S, T, \bigcup_{i=1}^{j} K_i, x, \widehat{\gamma}, \mu_j) \tag{2.10}$$

The extended definition 2.10 makes some changes to the entities in common with 2.8 from section 2.3. The unions $\bigcup_{i=1}^{j} N_i$ and $\bigcup_{i=1}^{j} E_i$ now belong to a subinterval of versions $[h, k]$ for the sequence of ICFGs. The symbol x is introduced for the same reason it was introduced in the extended definition of a CFG. A new entity $\bigcup_{i=1}^{j} K_i$ is added from the extended definition of a CFG with a subtle difference. The symbol y in the pairs $(c, y) \in \bigcup_{i=1}^{j} K_i$ are not used to reference a CFG, but each symbol is used to reference a MVCFG that represents the δ-chain of the CFG being called. The call node mapping function $\widehat{\gamma}$ is similar to the call node mapping function γ for a CFG with a key difference. Call nodes are mapped to other MVCFGs instead of the CFG for the procedure called.

The MVICFG $\mathcal{MV} = \{\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_n\}$ is a set of MVCFGs created for a sequence of ICFGs in the sequence $\mathcal{I} =< IC_1, IC_2, ..., IC_v >$ where n is the number of all possible δ-chains such that $n = |\mathcal{D}(\mathcal{I})|$. The MVICFG is a set of MVCFGs used to represent both intraprocedural and interprocedural changes across a sequence of ICFGs $< IC_1, IC_2, ..., IC_v >$ for v number of versions. The MVICFG contains an MVCFG to represent all versions of any procedure that exists in a sequence of ICFG versions. The definitions in this section provide the elements needed to define the algorithm used to construct an MVICFG in the next section.

Chapter 3

Constructing the MVICFG

3.1 General Algorithm for Constructing the MVICFG

The algorithm defined in this section is an abstract definition for constructing the MVICFG using set operations. The control flow graphs are mapped between consecutive versions of interprocedural control flow graphs in a sequence of graph versions. A comparison routine is used to identify which procedures are common to a pair of consecutive program versions as well as identify any added or deleted procedures. Once the procedures between consecutive program versions are matched, the respective control flow graphs between ICFGs are mapped. Node mappings between consecutive versions of control flow graphs are created using a differencing algorithm on two procedures to find the longest common subsequence of instructions the nodes represent. The node mapping is used to iteratively add control flow changes for each new version using the union of node and edge sets along with a method for annotating edges with version numbers. The benefit of defining the algorithm in terms of set operations is that we limit the discussion to the concept of the MVICFG without including details about using diff information from source code repositories. The complications of using diff information from source code repositories will be discussed later in the next sections.[1]

[1] The general algorithm for constructing an MVICFG in this section is more abstract than the algorithm described in the publication. The general algorithm for constructing an MVICFG uses a similar approach in section 4.1 of the publication. A key difference between the general algorithm described in this section and the algorithm described in the publication is that the general algorithm does not use diff information from source code repositories. The the modified algorithm in later sections adapts the general algorithm to make use of diff information [14].

The algorithm in this section has two parts described in Algorithm 1 and 2. Algorithm 1 takes a sequence of program versions $\mathcal{R}_1$, $\mathcal{R}_2$,...,$\mathcal{R}_v$ and builds the MVICFG representing v number of versions. Algorithm 1 describes the iterative process used to incrementally build MVCFGs for all the procedures in the sequence of program versions. The algorithm also describes how MVCFGs are created for new procedures added in subsequent versions of a program. Algorithm 2 is a supporting algorithm for Algorithm 1 which describes how a longest common subsequence is used to create a node mapping and perform a union of node and edge sets between control flow graphs.

Algorithm 1 takes the first step for constructing an MVICFG on line 3 by building the ICFG for the first version of the procedural program $\mathcal{R}_1$ referred to as IC_1. Lines 4-6 describe how the MVICFG is initialized from $\mathcal{R}_1$ by iteratively adding an MVCFG for each CFG in IC_1. The **AddNewCFG** procedure of the algorithm adds a new CFG to an MVICFG on line 5 as defined on lines 21-30. This procedure is used with the initial version of the program $\mathcal{R}_1$ as well as with new CFGs introduced by subsequent program versions. The initial version of each MVCFG for each new CFG is created and added to the MVICFG on lines 21-30. Line 22 describes how the edge set for a new CFG is labeled using the edge labeling procedure of the algorithm **LabelEdges** which generates the initial edge labeling function α_1 defined in section 2.1.2 (see Appendix A for a definition of **LabelEdges**). Recall that the edge labeling function is defined as $\alpha_1[a] = \{1\}$. The edge labeling function α_1 is assigned as the initial edge labeling function μ_1 for the union of edge sets in the MVCFG on line 23. After the edge labeling function is created, a new MVCFG is created and added to the set of MVCFGs known as the MVICFG. The new MVCFG is also added to the mapping function Γ on line 25 which will be used to retrieve the MVCFG with the symbol x while iterating over subsequent versions of CFGs. The call node mapping function $\widehat{\gamma}$ is also created on line 24 and initialized to *NIL* for each call node in K_1 on lines 26-28. The call node set K and the call node mapping function $\widehat{\gamma}$ are put in a queue Q on line 29. The call nodes are mapped to their respective MVCFGs with $\widehat{\gamma}$ later by dequeuing Q after all the MVCFGs are created for $\mathcal{R}_1$.

```
Input  : Sequence of program versions $\mathcal{R}_1$, $\mathcal{R}_2$,...,$\mathcal{R}_v$
Output: $\mathcal{MV}$ the multi-version ICFG representing v number of revisions
$\mathcal{MV} \leftarrow \emptyset$
$Q \leftarrow \emptyset$
$IC_1 \leftarrow$ BuildICFG ($\mathcal{R}_1$)
foreach $(N, E, S, T, K, x, \gamma) \in IC_1$ do
    AddNewCFG ($\mathcal{MV}, \Gamma, 1, N, E, S, T, K, x, Q$)
end
for $i \leftarrow 1 \ldots v-1$ do
    $(\mathcal{R}', \mathcal{R}^+) \leftarrow$ CompareProc ($\mathcal{R}_i, \mathcal{R}_{i+1}$)
    foreach $(x, P_{i+1}) \in \mathcal{R}^+$ do
        $(N_{i+1}, E_{i+1}, S, T, K_{i+1}, x) \leftarrow$ BuildCFG ($x, P_{i+1}$)
        AddNewCFG ($\mathcal{MV}, \Gamma, i+1, N_{i+1}, E_{i+1}, S, T, K_{i+1}, x, Q$)
    end
    UpdateMVICFG ($\Gamma, Q, \mathcal{R}', i$)
    while $Q \neq \emptyset$ do
        $(K^+, \widehat{\gamma}) \leftarrow$ Dequeue ($Q$)
        foreach $(c, x) \in K^+$ do
            $\widehat{\gamma}[c] \leftarrow \Gamma[x]$
        end
    end
end
Procedure AddNewCFG ($\mathcal{MV}, \Gamma, i, N, E, S, T, K, x, Q$)
$\alpha_i \leftarrow$ LabelEdges ($E, i$)
$\mu_i \leftarrow \alpha_i$
$\mathcal{MV} \leftarrow \mathcal{MV} \cup \{(N, E, S, T, K, x, \widehat{\gamma}, \mu_i)\}$
$\Gamma[x] \leftarrow (N, E, S, T, K, x, \widehat{\gamma}, \mu_i)$
foreach $(c, x) \in K$ do
    $\widehat{\gamma}[c] \leftarrow NIL$
end
Enqueue ($Q, K, \widehat{\gamma}$)
End Procedure
```

Algorithm 1: General algorithm to build an MVICFG for a sequence of program versions $\mathcal{R}_1, ..., \mathcal{R}_v$

Once the MVICFG is initialized, Algorithm 1 begins iteratively building the MVICFG for the remaining sequence of program versions in $\mathcal{R}_2, ..., \mathcal{R}_v$ on lines 7-19. Each consecutive pair of program versions $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ are compared with **CampareProc** on line 8 to identify both new procedures and common procedures (see Appendix A for a definition of **CompareProc**). The algorithm procedure **CompareProc** returns a pair of sets of procedures $(\mathcal{R}', \mathcal{R}^+)$. The set $\mathcal{R}'$ contains a 3-tuple for each of the procedures that are common to both program versions $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ such that $(x, P_i, P_{i+1}) \in \mathcal{R}'$ when $(x, P_i) \in \mathcal{R}_i$ and $(x, P_{i+1}) \in \mathcal{R}_{i+1}$. The tuples in $\mathcal{R}'$ are used to create node mappings between two versions of a procedure using a differencing algorithm. The set $\mathcal{R}^+$ contains the new procedures added in version $\mathcal{R}_{i+1}$ such that $(x, P_{i+1}) \in \mathcal{R}^+$ when $(x, P_{i+1}) \in \mathcal{R}_{i+1}$ but $(x, P_i) \notin \mathcal{R}_i$. New procedures are processed first on lines 9-12 to ensure MVCFGs for new procedures are mapped with Γ before any call node mappings are created. A control flow graph is created for each new procedure in $\mathcal{R}^+$ on line 10 using **BuildCFG**. The algorithm procedure **AddNewCFG** that was used to initialize the MVICFG is used on line 11 to add an initial MVCFG for each new procedure discovered in each successive version. Once the set of new procedures are exhausted for version i, $\mathcal{R}'$ is processed using Algorithm 2 in **UpdateMVICFG** on line 13.

Algorithm 2 accepts four arguments to process the procedures that are common to versions $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$. The set $\mathcal{R}'$ contains the procedures common to both versions which will be compared for changes. The mapping function Γ is used to retrieve the MVCFG created in previous versions for each procedure in $\mathcal{R}'$. The queue used to add new call nodes introduced by P_{i+1} is passed as the argument Q. The fourth argument is the current version number i used to label edges for the next iteration of the MVICFG.

The procedures in $\mathcal{R}'$ are processed on lines 1-13 of Algorithm 2 where changes for each procedure are incrementally added to each corresponding MVCFG that will represent control flow changes between $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ for all procedures common to both versions. The first step is to find the longest common subsequence L of instructions for P_i and P_{i+1} using the **GenerateLCS** procedure on line 2. Any of the common algorithms for finding

```
   Input  : a list of procedures $\mathcal{R}'$ , $\Gamma$ mapping of MVCFGs, $Q$ the queue for new call nodes ,
            and $i$ for the current version
   Output: $\mathcal{MV}$ the updated multi-version ICFG, $Q$ the updated queue with new cal nodes
1  Procedure UpdateMVICFG ($\Gamma, Q, \mathcal{R}', i$)
2  foreach $(x, P_i, P_{i+1}) \in \mathcal{R}'$ do
3      $L \leftarrow$ GenerateLCS $(P_i, P_{i+1})$
4      $\Phi_i \leftarrow$ CreateNodeMapping $(L, P_i, P_{i+1})$
5      $(N_{i+1}, E_{i+1}, S, T, K_{i+1}, x) \leftarrow$ BuildCFG $(x, P_{i+1})$
6      $\alpha_{i+1} \leftarrow$ LabelEdges $(E_{i+1}, i+1)$
7      $(N, E, S, T, K, x, \widehat{\gamma}, \mu_i) \leftarrow \Gamma[x]$
8      UnionNodes $(N, N_{i+1}, \Phi_i)$
9      UnionCallNodes $(K, K_{i+1}, \Phi_i)$
10     UnionEdges $(E, E_{i+1}, \Phi_i, \mu_i, \alpha_{i+1})$
11     $K^-_{i+1} \leftarrow$ IntersectCallNodes $(K, K_{i+1}, \Phi_i)$
12     $K^+_{i+1} \leftarrow K_{i+1} - K^-_{i+1}$
13     Enqueue $(Q, K^+_{i+1}, \widehat{\gamma})$
14 end
15 End Procedure
```

Algorithm 2: UpdateMVICFG procedure supports Algorithm 1 by incrementally updating MVCFGs for procedures common to both program versions $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$.

a longest common subsequence such as Hirschberg's algorithm used in the UNIX diff tool can be used [10]. The longest common subsequence L is used to generate the node mapping Φ_i on line 3 necessary to perform a union of node and edge sets. See section 2.2 for more details about the node mapping function Φ_i.

The next lines 4-9 update the MVCFG for P_i and P_{i+1} by creating a new CFG and performing a union with MVCFG of previous versions. Once the node mapping Φ_i is created, a CFG is created for the new procedure P_{i+1} on line 4 of Algorithm 2. The edge set E_{i+1} for the new CFG is labelled on line 5, and the edge labelling function α_{i+1} is created. The next step is to retrieve the MVCFG from previous iterations using Γ on line 6 and add the changes from P_{i+1} . The sets $N, E,$ and K are the unions from previous versions where t is the number of previous versions, $N = \bigcup_{k=i-t+1}^{i} N_k$, $E = \bigcup_{k=i-t+1}^{i} E_k$, and $K = \bigcup_{k=i-t+1}^{i} K_k$. The node and edge sets N_{i+1}, E_{i+1} and K_{i+1} in line 4 from the CFG of P_{i+1} are joined in union with N, E and K respectively from line 6 on lines 7-9. The union operations all require the use of the node mapping function Φ_i. The union of the node sets

N and N_{i+1} is done on line 7, and the union of call node sets K and K_{i+1} is done on line 8. The union of edge sets E and E_{i+1} is done on line 9 with **UnionEdges** which also updates the edge labeling function μ. The **UnionEdges** procedure uses the node mapping function Φ_i to map edges between E and E_{i+1} and updates μ by union of the labels from α_{i+1}.

The last steps for Algorithm 2 select all the new call nodes added in P_{i+1} and place them in the queue Q with the call node mapping function $\widehat{\gamma}$ for updating the call flow in the MVICFG. This queue Q will be process in the final steps in the iteration of $\mathcal{R}_2, .., \mathcal{R}_v$ in Algorithm 1. The intersection between the call node sets K and K_{i+1} is returned by **IntersectCallNodes** on line 10 which uses the node mapping function Φ_i. The set of intersecting nodes K^-_{i+1} is then removed from K_{i+1} to produce the set of new call nodes K^+_{i+1} introduced in P_{i+1}. The last step on line 12 puts the set of new nodes K^+_{i+1} and the call node mapping function in Q.

Once Algorithm 2 finishes processing all the procedures common to the consecutive program versions $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$, the final steps of an iteration will execute on lines 14-18 in Algorithm 1. The queue of new call nodes Q and the mapping Γ will be used to update the call node mapping functions $\widehat{\gamma}$ with the changes in the call flow represented by the MVICFG. If Q is not empty, the pairs of new call nodes and related call mapping function $(K^+, \widehat{\gamma})$ are dequeued on line 15. Each call node and symbol pair $(c, x) \in K^+$ is used with Γ to lookup the MVCFG for the called procedure and map it to the call node c with the call mapping function $\widehat{\gamma}$. On line 17, the MVCFG is returned by $\Gamma[x]$ and stored in $\widehat{\gamma}[c]$. All of the new call nodes added from changes in $\mathcal{R}_{i+1}$ will be mapped to a MVCFG once Q is exhausted.

The complete MVICFG is constructed once the algorithm completely iterates over all of the program versions in the sequence $\mathcal{R}_1, ..., \mathcal{R}_v$. Algorithms 1 and 2 describe the basic steps needed to construct an MVICFG using the definitions presented in chapter 2. The general algorithm represents a simple form for the algorithm used to construct MVICFGs in this study. However, the most significant drawback of this algorithm is that it is inefficient. A significant inefficiency is that the differencing algorithm and union operations are performed on every version of every procedure. Since a typical revision history contains a small percent change between revisions of a program, the efficiency of the construction algorithm would be greatly increased by limiting the use of Algorithm 2 to the procedures that have changed (see chapter 4 for data about change in revision histories). The algorithm would also reduce the storage demands of the MVICFG by using a different scheme for labeling edges with version numbers.

The next sections describe how the general algorithm is modified to increase the algorithm efficiency and reduce the storage demands for constructing the MVICFG. The modified algorithm uses data mined from a software revision history to identify which procedures change between revisions and collect diff information by comparing source code. The modified algorithm also uses a different scheme for labeling edges in an MVCFG. The modified algorithm is the algorithm implemented to produce the results of the study presented in chapter 4.

3.2 The Lazy Labelling Method

The modified algorithm used to construct a MVICFG uses an edge labelling method called lazy labelling which reduces the storage demands of the MVICFG. The labelling scheme used in the previous sections is the same as the method used with Multi-Version Graphs defined in section 2.1. Labelling edges with sets of discrete version numbers is not an efficient method for representing a sequence of control flow graphs. The edges in a MVCFG always belong to a subsequence of control flow graphs for an interval of versions in a revision history. Listing intermediate version numbers in a discrete set for a subsequence of graphs does not provide anymore information than simply using a pair of numbers to identify an interval of versions for the control flow edge. Thus, an interval annotation for edge labelling in a MVCFG is more efficient.

The lazy labelling method is as conservative as possible by only labelling enough edges to determine the versions for all edges and nodes in an MVICFG. Using intervals to label every edge in a MVICFG is not the most conservative approach. There are other opportunities for optimizing edge labels. Some edges don't need to be labelled, especially edges that belong to every version of an MVCFG. The set of all version numbers for an MVICFG can be replaced by the empty set $\emptyset$ to reduce storage demands. There are no storage demands for the $\emptyset$ because it contains no elements. The $\emptyset$ label can also be used to label edges when the version interval for an edge can be inferred by incident edges. Path information can be used in a control flow graph to determine the version interval of an edge. Path sensitive analysis can be used to determine versions because all paths begin at the start node S and end at the end node T in a control flow graph. The only edges that are important are the ones incident with nodes where paths diverge or converge across versions. The nodes used to mark the divergence or convergence of paths across versions are called v-branch and v-merge nodes.

Figure 3.1 shows an example of a Multi-Version Graph that uses the lazy labelling method. The graph G_1 is the first graph in a sequence of three graphs $< G_1, G_2, G_3 >$. The three boxes in Figure 3.1 represent the Multi-Version Graphs for G_1, $G_1 \cup G_2$, and $G_1 \cup G_2 \cup G_3$. Each box represents the Multi-Version Graph as it is constructed incrementally with each union operation. This illustration shows how edge labels are updated with changes from each graph so that each graph in the sequence is represented correctly.

The first box in Figure 3.1 shows the Multi-Version Graph for only one graph G_1 in the sequence $< G_1, G_2, G_3 >$. No labels are required for a Multi-Version Graph that represents a single graph in a sequence. Thus, all edges in the Multi-Version Graph for G_1 are labelled with $\emptyset$ since all the nodes and edges belong to G_1 in the first increment. Consequently, any Multi-Version Graph used to represent one version of a graph does not demand anymore storage capacity than the graph represented. The Multi-Version Graph for G_1 contains no v-branch and v-merge nodes, and it is used as the baseline graph for the subsequent union operations.

The edges in $G_1 \cup G_2$ in Figure 3.1 are labelled since there are changes between G_1 and G_2. Node 2 becomes a v-branch node and node 5 becomes a v-merge node. The edge

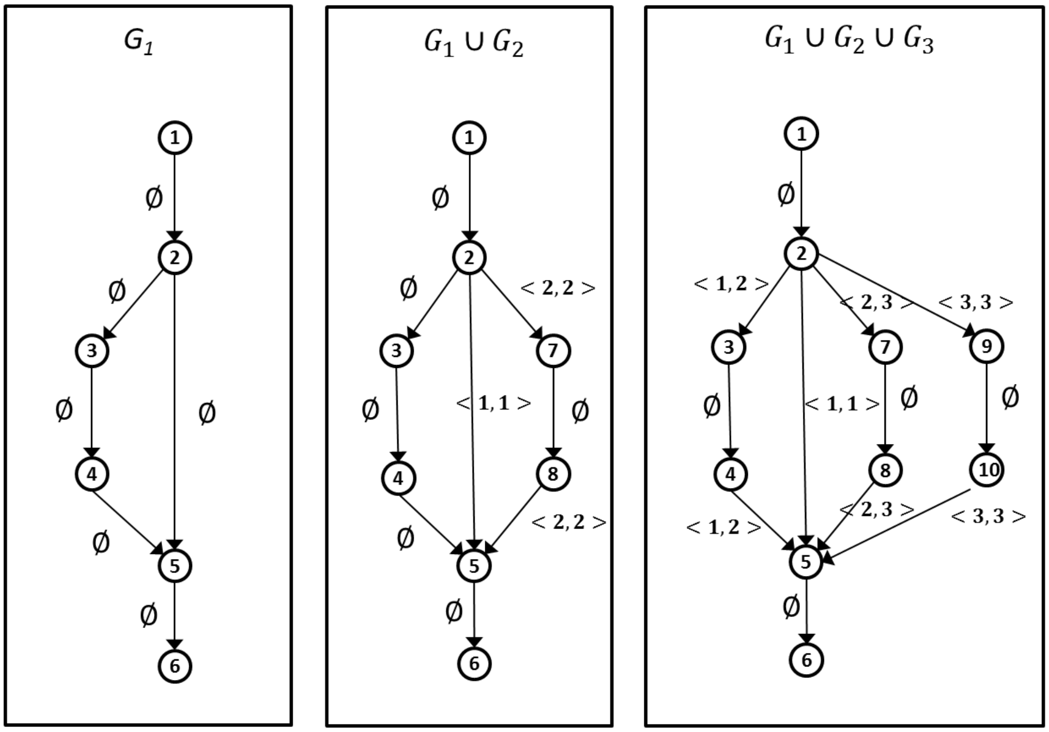


Figure 3.1: The illustration shows how the lazy labeling method works with the union of the three graphs G_1, G_2 and G_3. Node 2 is a v-branch node, and node 5 is v-merge node. Only the edges that are adjacent to v-branch and v-merge nodes are labeled.

$(2, 5)$ incident with both v-merge and v-branch nodes is labelled with $< 1, 1 >$ since it only belongs to G_1. The edges $(8, 5)$ and $(2, 7)$ are new edges in G_2 so they are labelled with $< 2, 2 >$. The edge $(2, 3)$ belongs to both G_1 and G_2 so it is labelled with $\emptyset$ as belonging to all versions. Likewise, the edge $(4, 5)$ incident with the v-merge node is labelled with $\emptyset$ since it belongs to both graphs.

Not all edges labelled with $\emptyset$ belong to all versions. The edge $(7, 8)$ is a new edge in G_2, but it is still labelled with $\emptyset$. In the lazy labelling scheme, the $\emptyset$ label is used when an explicit label is unnecessary. The version of $(7, 8)$ would be implied by the versions of the paths incident with the edge. Since the incident edge $(2, 7)$ is labelled with $< 2, 2 >$, all paths that include the edge $(7, 8)$ belong to version 2. Thus, the version of $(7, 8)$ is implicit.

The labels of the edges incident with the v-merge and v-branch nodes are updated as new edges and nodes are added in $G_1 \cup G_2 \cup G_3$. The label for edge $(2, 3)$ is changed from $\emptyset$ to $< 1, 2 >$ because it does not belong to G_3. Likewise, the label for edge $(4, 5)$ is changed from $\emptyset$ to $< 1, 2 >$. The labels for edges $(2, 7)$ and $(8, 5)$ must be updated because they belong to both G_2 and G_3. Thus, the labels for these edges change from $< 2, 2 >$ to $< 2, 3 >$. The new edges $(2, 9)$ and $(10, 5)$ are labelled with $< 3, 3 >$ because they are incident to v-branch and v-merge nodes. The edge $(9, 10)$ is not labelled because it is not necessary.

Any of the graphs G_1,G_2, or G_3 can be reconstructed from the Multi-Version Graph $G_1 \cup G_2 \cup G_3$ using a depth first search starting at node 1. For example, the node and edge sets N_3 and E_3 for graph G_3 can be reconstructed as follows. The node 1 is the starting node so it gets added to N_3. There is only one incident edge $(1, 2)$ labeled with $\emptyset$ so it gets added to E_3. Now we visit node 2 and add it to N_3. Next the labels for the incident edges $(2, 3)$, $(2, 7)$, and $(2, 9)$ are checked. The edge $(2, 3)$ is labelled with $< 1, 2 >$ so it is discarded. The edges $(2, 7)$ and $(2, 9)$ are labelled $< 2, 3 >$ and $< 3, 3 >$ respectively so they are added to E_3. Next we visit node 7 and add it to N_3 and check the incident edge $(7, 8)$. The edge $(7, 8)$ is labelled with $\emptyset$ so it is included. The node 8 is visited and added to N_3. Edge $(8, 5)$ is labelled with $< 2, 3 >$ so it gets included and node 5 is visited. From here the edges $(5, 6)$,$(9, 10)$,and $(10, 5)$ are added respectively along with the incident nodes $6, 9$, and 10. Completing the search we have $N_3 = \{1, 2, 7, 8, 5, 6, 9, 10\}$ and $E_3 = \{(1, 2), (7, 8), (8, 5), (5, 6), (2, 9), (9, 10), (10, 5)\}$. [2]

The lazy labelling method provides an effective method for reducing the storage demands of the MVICFG. However, the lazy labelling method negatively impacts the computational complexity of the modified algorithm used to construct the MVICFG. The algorithm must do extra work when creating a v-branch or v-merge nodes in the union operation

[2] One might point out that the information from the v-merge node in the previous example was not necessary to reconstruct G_3 using a depth first search beginning at the starting node. However, a MVICFG is constructed to facilitate path sensitive analysis for change verification across multiple versions of a program. A path sensitive analysis such as demand driven analysis raises queries at intermediate nodes of interest and traverses the graph counter to the control flow direction in the analysis. Thus, v-merge nodes ensure that a counter directional walk of the control flow graph is guided along the appropriate paths for the versions under analysis.

```
Input  : $N, E, E_{i+1}, \mu, \Phi$
Output: Updated edge set $E$ and labeling function $\mu$
foreach $a \in E_{i+1}$ do
    if $a \notin_\Phi E$ then
        $(n_1, n_2) \leftarrow a$
        $\mu[a] \leftarrow < i+1, i+1 >$
        if $n_1 \in_\Phi N \vee n_2 \in_\Phi N$ then
            MakeVersionNode $(n_1, n_2, N, E, E_{i+1}, \mu, \Phi)$
        end
        else
            $\mu[a] \leftarrow \emptyset$
        end
        $E \leftarrow E \cup \{a\}$
    end
    else
        $v \leftarrow \mu[a]$
        if $v \neq \emptyset$ then
            $< i_1, i > \leftarrow v$
            $\mu[a] \leftarrow < i_1, i+1 >$
        end
    end
end
```

Algorithm 3: This algorithm performs the union of edge sets for a MVCFG using the lazy labeling method.

of the edge sets. Incident nodes and edges must be scanned for each new edge in a union operation to update the version interval in the labels. The Algorithms 3 and 4 illustrate the trade-off with computational efficiency by defining the algorithm for the lazy labelling method used with the union operation for edge sets.

The algorithm defined Algorithm 3 shows how the union operation would be performed on the edge sets for a CFG and a MVCFG. Algorithm 3 takes the inputs N, E, E_{i+1}, μ, and Φ where (1) $N = \bigcup\limits_{k=j}^{i} N_k$ is the set of nodes and $E = \bigcup\limits_{k=j}^{i} E_k$ is the set of edges in the MVCFG, (2) μ denotes the labelling function μ_i from the MVCFG, (3) E_{i+1} is the edge set for the next CFG in the union of the sequence of versions, and (4) Φ is the node mapping function for versions i to $i+1$.. The output of Algorithm 3 is an updated edge set E and labelling function μ with changes from the edge set E_{i+1}. Algorithm 4 defines the procedure **MakeVersionNode** that supports Algorithm 3 by creating v-branch and v-merge nodes.

The Algorithm 3 starts by iterating over each edge in the edge set E_{i+1} for the next version of the CFG in the union of a sequence of CFGs. Each edge $a \in E_{i+1}$ is checked on line 2 to find new edges from changes in version i+1. The edge set E from the MVCFG is used to search for new edges in $a \notin_\Phi E$. The $\notin_\Phi$ operator uses the node mapping function Φ to check that the edge $a \in E_{i+1}$ does not exist in E.[3] If the edge is not new, the edge tag is updated on lines 14-18. The edge label is only updated when the $\emptyset$ label was not used in version i. Thus, either the edge a is not incident to a v-merge or v-branch node, or the edge belongs to all versions. Otherwise, the edge label $< i_1, i >$ is updated to include the next version with the label $< i_1, i+1 >$ (see Appendix D for proof).

The lines 3-11 of Algorithm 3 add new edges from E_{i+1} in each iteration where a new edge is found. Each new edge is initially labelled with $< i+1, i+1 >$ as it would only belong to version $i+1$. The new edge must be checked on line 5 to find any incident nodes that map to previous versions. If the new edge is not incident with any nodes that map to previous versions, then the $\emptyset$ label is used instead of $< i+1, i+1 >$. Otherwise, a v-branch or v-merge node must be created using the incident nodes. The v-branch node n_1 and v-merge node n_2 candidates are passed to the **MakeVersionNode** procedure defined in Algorithm 4. At least one, if not both, of the nodes n_1 and n_2 will become incident to labelled edges if not already. A new edge $a \in E_{i+1}$ is added to E once labelled correctly and after the appropriate v-branch and v-merge nodes are created.

[3] Given a Φ_i relation defined in the definitions of section 2.2, the $\in_\Phi$ operator is defined for node sets and edge sets as follows. Given the node sets N, N_i, and N_{i+1} where $N_i \subseteq N$, let n be a node such that $n \in N_i$. We say that $n \in_\Phi N$ *iff* $\exists m \in N_{i+1}$ *s.t.* $\Phi_i(nl(n)) = nl(m))$. Furthermore, given the edge sets E, E_i, and E_{i+1} where $E_i \subseteq E$, let $a = (n_1, n_2)$ be an edge such that $a \in E_i$. We say that $a \in_\Phi E$ *iff* $\exists a' \in E_{i+1}$ where $a' = (m_1, m_2)$ *s.t.* $\Phi_i(nl(n_1)) = nl(m_1))$ and $\Phi_i(nl(n_2)) = nl(m_2))$.

```
Procedure MakeVersionNode ($n_1, n_2, N, E, E_{i+1}, \mu, \Phi$)
if $n_1 \in_\Phi N$ then
    $A \leftarrow$ AdjOut ($n_1, E, \Phi$)
    foreach $adj \in A$ do
        if $adj \notin_\Phi E_{i+1}$ then
            $v \leftarrow \mu[adj]$
            if $v = \emptyset$ then
                $i_1 \leftarrow$ FindMinVersion ($n_1, E, \mu$)
                $\mu[adj] \leftarrow < i_1, i >$
            end
        end
    end
end
if $n_2 \in_\Phi N$ then
    $A \leftarrow$ AdjIn ($n_2, E, \Phi$)
    foreach $adj \in A$ do
        if $adj \notin_\Phi E_{i+1}$ then
            $v \leftarrow \mu[adj]$
            if $v = \emptyset$ then
                $(m_1, n_2) \leftarrow adj$
                $i_1 \leftarrow$ FindMinVersion ($m_1, E, \mu$)
                $\mu[adj] \leftarrow < i_1, i >$
            end
        end
    end
end
End Procedure
```

Algorithm 4: MakeVersionNode procedure used in Algorithm 3 adds v-branch or v-merge nodes when a new edge from E_{i+1} is incident with any nodes that map to previous versions.

The Algorithm 4 is responsible for creating a v-branch node or v-merge node when n_1 or n_2 map to a node in N. A v-branch node is created on lines 1-13 and a v-merge node is created on lines 14-26. The node n_1 is checked on line 2 and the node n_2 is checked on line 14 to detect a mapping to a node from a previous version. Either one or both of the nodes are used to create a v-branch or v-merge node.

When n_1 is determined to be a v-branch node, a set of incident edges directed out from node $n_1 \in_\Phi N$ is collected from the MVCFG edge set E using the **AdjOut** procedure on line 3. The version label for each incident edge will not be updated if the edge is common to versions i and $i+1$. The edge is checked on line 5 to determine if the edge was removed in version $i+1$. If the edge is determined to be removed, the edge label must exclude the version number $i+1$. If version label exists for the removed edge that is an interval and not the $\emptyset$ label, than it should already have a label of the form $< i_1, i >$. Otherwise, an explicit edge label must be created for the removed edge. The starting version for the new edge label is searched for using the **FindMinVersion** procedure. This procedure does a depth first search opposite of the control flow to check all paths to the node n_1 and determines the minimum version where the node is reachable. The minimum version is denoted as i_1, and it is used to create the version interval $< i_1, i >$ used to label the incident edge.

When n_2 is determined to be a v-merge node, the algorithm follows steps similar to those for creating a v-branch node. The set of edges incident to n_2 are labelled just like the edges incident to a v-branch node. However, the set of incident edges are directed in toward node $n_2 \in_\Phi N$ and collected from the MVCFG edge set E using the **AdjIn** procedure on line 15. Only the incident edges that have $\emptyset$ as a label are updated with a version range on line 22. The minimum version used to label an edge for a v-merge node is found using the adjacent node incident of the edge being labelled.

Elements of the Algorithms 4 are used to create v-branch and v-merge nodes in the modified algorithm in section 3.3. Section 3.3 will show how information mined from source code repositories is used with the lazy labelling method to improve the efficiency of the algorithm used to construct a MVICFG. Although the lazy labelling method is a trade-off with computational complexity, the scalability of the algorithm is improved by reducing the storage demands of the MVICFG. The use of source changes mined from source code repositories help offset the increase in computational complexity.

3.3 Modified Algorithm for Constructing the MVICFG

The modified algorithm for constructing the MVICFG is used to incrementally build a graph representation of the interprocedural control flow for all revisions of a program in a software revision history. The algorithm leverages the differential information mined from software repositories where textual differences are used in the incremental build of the MVICFG. The differential information in this study takes the form of UNIX style diffs generated by file comparisons using Hircshberg's LCS algorithm [10]. The UNIX diffs are units of code change consisting of lines of code that were inserted or deleted at a file location in a single revision. The textual differences from source code files are used by the modified algorithm to identify only those functions that have changed and limit the incremental updates to only those MVCFGs that correspond to changed functions. This can dramatically improve the performance of the algorithm by reducing the number of union operations needed to construct the MVICFG. This is especially true for typical software revision histories since the changes between revisions tend to impact a small percentage of code local to only a few procedures.

The modified algorithm translates textual changes in source code into graph changes in the control flow representation of a program. The problem with using textual changes in source code is that not all changes effect control flow changes in a program. Changes in source might only effect changes in white space or source code comments. Changes might also be structural such as reordering class functions or variable declarations. Other changes might show up that are only lexical such as variable name changes, namespace changes, class name changes, new macro statements, etc. Even a function name or parameter name change can be construed as an interprocederal change. A method is needed to filter out the

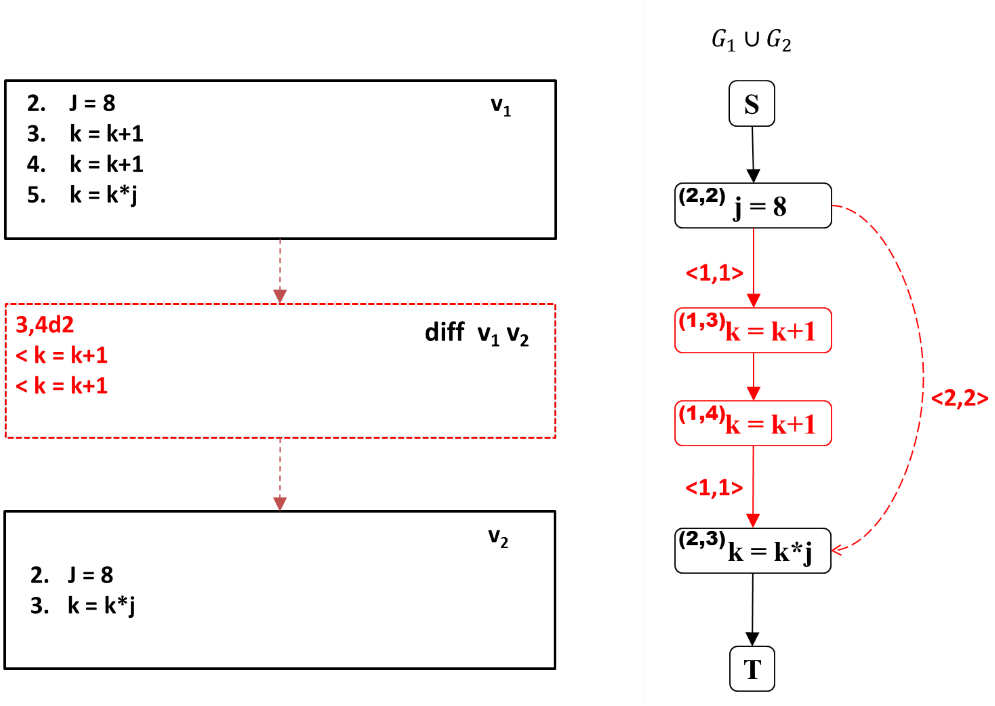


Figure 3.2: The graph diff shows how the nodes corresponding to deleted statements are not incident with all the edges related to the diff. The edge $(2,2) \rightarrow (2,3)$ is not incident with any of the nodes deleted on lines 3 and 4 of v_1.

information in source code files that are not relevant to the semantics of a program.

Translation of textual changes in source code into graph changes in the control flow representation of a program must overcome other challenges besides filtering out irrelevant lexical changes. Currently, there is no known method for translating code fragments from code changes into the control flow edges corresponding to control flow changes related to the code changes. Mapping code change to the corresponding added or deleted nodes in subsequent control flow graphs is less problematic. Once the non-essential lexical changes are filtered out, mapping program statements to control flow nodes is relatively straight forward. Identifying the control flow edges that complement the intersection of edge sets of subsequent control flow graphs which represent the control flow change is not straight forward. The context of the control flow graph node changes becomes important. For instance, the nodes corresponding to deleted statements in a program are often not incident with the new control edges introduced by the deletion. The new control flow edges related

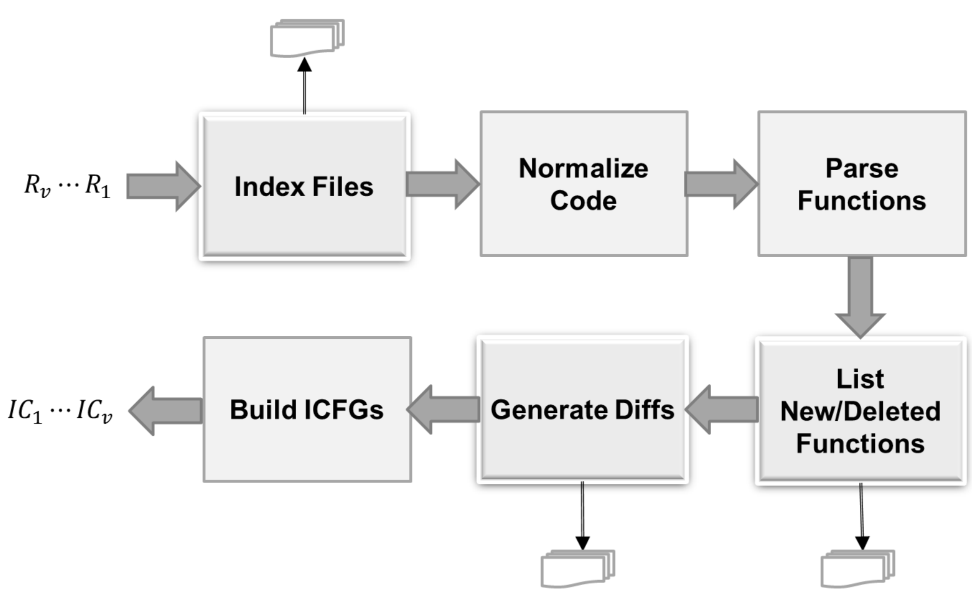


Figure 3.3: The steps in the preprocessing stage for building an MVICFG from source code mined from source code repository. The source code $R_1..R_v$ for v number of versions is processed to generate diff information and the ICFGs $IC_1..IC_v$ used to construct the MVICFG.

to deleted program statements are often incident to nodes adjacent to the deleted nodes. Consequently, the unchanged statements of a program outside the regions of code change become important when constructing a MVICFG. See Figure 3.2

A preprocessing phase is used to distill the information mined from source code repositories into the information needed to construct the MVICFG. Figure 3.3 illustrates the six major steps in the preprocessing phase for constructing the MVICFG. The preprocessing phase is executed over the entire range of revisions represented by the MVICFG. The sequence $< R_1..R_v >$ in Figure 3.3 represents the sequence of whole programs pulled from a source code repository where v is the number of program revisions being represented. Four sets of artifacts are created to support the construction of the MVICFG when the preprocessing phase is completed. While the definition of the modified algorithm does not explicitly depend on a preprocessing phase, the artifacts created by the preprocessing phased are used implicitly in the subprocedures that support the modified algorithm.

The first stage of the preprocessing phase creates an index of all the source code files that belong to each revision of a program. The list of file names for each program revision include all changed and unchanged files of the program. These file names are used in the modified algorithm to group together functions with their respective textual diff information. The grouping of functions by filename is important when creating the correct node maps for subsequent CFGs as we will see later in the definition of the algorithm.

The next stage of the preprocessing phase shown in Figure 3.3 is the **Normalize Code** stage. This stage takes the list of all code files for each program revision and runs each code file through a code formatter and code preprocessor. The code formatter ensures that only one program statement appears on a line for each file [4] The C compiler preprocessor is also used in this stage to handle macro substitution for C/C++ programs. The preprocessor ensures that changes in macro definitions are reflected in the function definitions before the diff information for each revision is generated. The Normalize Code stage helps filter out some of the previously mentioned lexical changes.

Once all of the source code files have been normalized, the next stage of preprocessing parses the code in the bodies of the function definitions for comparison between program revisions. The parse function stage indexes all the function names for each program revision, and the file name and line number for each function definition is added to a function index. The code in the function bodies are copied to separate files to be compared using the UNIX diff tool to generate diff information in the next stages in preprocessing.

The fourth stage in the preprocessing phase is the **List New/Deleted Functions** stage. This stage does a lexicographical comparison of function names between successive versions of a program. The list of new and deleted functions is created for each revision number to support the incremental interprocedural updates for the MVICFG in the modified algorithm. This list is used by the algorithm to determine when to add a new MVCFG for a new function created in a revision of a program.

[4] For the sake of full disclosure, the code formatter was not used in the results of this study because it was not needed for the benchmarks selected. It is a common coding practice to put a line break following each statement in a program. While some statements will span multiple lines, the algorithm implementation in this study includes some heuristics to select all the correct nodes for statements that span multiple lines. The heuristics used are not included in the algorithm definition.

The fifth stage of preprocessing generates the textual diff information used by the modified algorithm to incrementally update MVCFGs with graph changes for each program revision. The textual diff information is generated by performing a UNIX style diff on the normalized code for the function bodies generated by stages two and three. A list of diffs is created for each program revision and grouped by function name. The resulting diff information will be cleaner than the diff information generated by source code repositories such as SVN or Git. Source code comments and white space changes will have been filtered out by the code normalization stage. The structural diff information will be filtered out by extracting the code for function bodies. The remaining code changes will more closely approximate control flow changes in a program revision.

The final stage of the preprocessing phase for constructing an MVICFG is the **Build ICFGs** stage. This stage generates the sequence of interprocedural control flow graphs for the sequence of program revisions in a revision history. All the control flow graphs for each program revision are created and stored as an artifact used by the modified algorithm to build the MVICFG. The complete ICFGs for each revised program is not necessary. Only the complete ICFG for the first program revision R_1 is needed as the baseline MVICFG. Partial ICFGs are generated for the remaining program revisions in the sequence $< R_2, ..., R_v >$ which only contain CFGs for new and modified functions.

Figure 3.4 illustrates how graph diffs are connected to the baseline MVCFG from incremental updates using textual diffs. The graphs v_1, v_2, and v_3 represent three consecutive CFGs generated in the preprocessing phase which correspond to a modified function in three successive program revisions. The CFG for v_1 represents the first version of a function so it becomes the baseline graph for the MVCFG. The textual diffs are used to identify the line numbers of the source code revisions, and nodes that added or deleted in subsequent revisions are selected using these line numbers. For example, the program revision v_2 inserts three new lines of code on line 5 of the program. Thus, the nodes labelled (2.5), $(3, 5)$, and $(3, 6)$ are connected to the baseline MVCFG using v-branch node (2.4) and v-merge node $(3, 9)$. The algorithm creates v-branch and v-merge nodes by mapping nodes adjacent

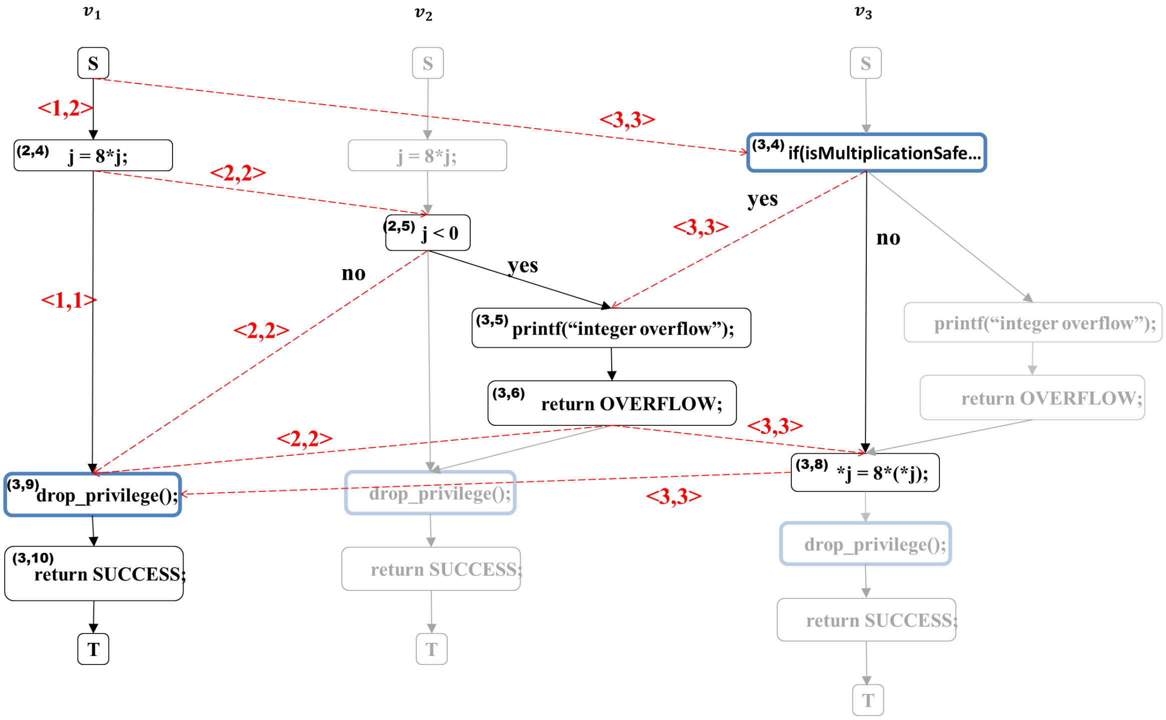


Figure 3.4: The modified algorithm constructs an MVCFG incrementally by extracting nodes for successive CFGs and joining them to the baseline MVICFG via edges connecting v-branch and v-merge nodes. The red edges show how successive CFGs are joined using graph diffs.

to inserted nodes to the baseline MVCFG from the previous revision. The red line in Figure 3.4 shows the new edges that are incident with the v-branch and v-merge nodes.

Figure 3.4 also shows how nodes are labelled using line numbers in order to map nodes between versions. While every node in an MVCFG corresponds to a specific line number in the source code for the statement the node represents, a line number is not enough to accurately identify node in the node set in an MVCFG. When a node is created to represent a statement in a program, it is labelled with the line number from the first version of the program that introduced the statement. As lines of code are added and deleted to successive versions of a program, the line number for the corresponding statement will change. Thus, the node label needs to be updated when the MVCFG is updated with changes in each new version.

Another problem occurs with line numbers when statements are added and deleted on the same line number across multiple versions. For example, in Figure 3.4 the node for `j = 8*j;` represents a statement in v_1 and the node for `if(isMultiplicationSafe...)` represents a different statement in v_3 both of which are on line number 4. Simply labelling both nodes with the line number 4 would be ambiguous and potentially cause incorrect node maps during an incremental build of the MVCFG. The revision number is paired with line numbers to keep track of the last program version the node belongs. For instance, all of the nodes $(3,4),(3,5),(3.6),(3.8).(3.9)$, and $(3,10)$ are current as of version 3. The node map for the next set of updates from the program revision 4 will only map to nodes with line numbers from version 3. Thus, the nodes $(2,4)$ and $(2,5)$ will be ignored in all revisions beyond version 3 when building the MVCFG.

The modified algorithm builds on the gereral algorithm defined in Algorithm 1 and 2 in section 3.1 by introducing the optimizations previously discussed. Elements of the lazy labelling algorithm are used in the modified algorithm to improve the storage efficiency of the MVICFG. The computational complexity of the algorithm is improved by distilling the information from source code repositories using a preprocessing algorithm. The artifacts gererated in the preprocessing phase are used to incrementally build the MVICFG for a sequence of program versions with fewer union operations than the general algorithm. The Algorithms 5, 6, 7. and 8 provide a complete definition of the algorithm for constructing the MVICFG to represent a sequence of interprocedural control flow graphs from a software revision history.

The Algorithm 5 is the high level algorithm that builds an initial version of an MVICFG for the original version of a program v_1, and then incrementally updates it with changes for each revision number $v_2, ..., v_n$. The interprocedural control flow graph IC_1 on line 3 is used for the baseline MVICFG, and the initialization of $\mathcal{MV}$ is performed by the loop on lines 4-6. The **AddNewCFG** procedure adds a new MVCFG to $\mathcal{MV}$ as it was defined in Algorithm 1 of the general algorithm but with the lazy labelling method. Thus, all the control flow edges in $\mathcal{MV}$ are labelled with $\emptyset$ instead of v_1.

```
Input  : $V$, set of revision numbers of source code $v_1$, $v_2$, ... ,$v_n$
Output: $MV$, the MVICFG representing n revisions
1  $\mathcal{MV} \leftarrow \emptyset$
2  $Q \leftarrow \emptyset$
3  $IC_1 \leftarrow$ BuildICFG ($v_1$)
4  foreach $C \in IC_1$ do
5      AddNewCFG ($\mathcal{MV}, C, v_1, \Gamma, Q$)
6  end
7  foreach $v_i, v_{i+1} \in V$ do
8      FuncNames = GetNewFunctions ($v_{i+1}$)
9      foreach $FuncName \in FuncNames$ do
10         $C \leftarrow$ BuildCFG ($FuncName, v_{i+1}$)
11         AddNewCFG ($\mathcal{MV}, C, v_{i+1}, \Gamma, Q$)
12     end
13     ChangedFiles = GetDiffFiles ($v_i, v_{i+1}$)
14     foreach $FileName \in ChangedFiles$ do
15         UpdateMVCFG ($FileName, v_{i+1}, \Gamma, Q$)
16     end
17     UpdateNewCallNodes ($\Gamma$,$Q$)
18 end
```

Algorithm 5: The modified algorithm builds an initial version of the MVICFG for v_1, and then incrementally updates it with changes for each revision numbers $v_2, ..., v_n$.

Each revision number $v_i \in V$ corresponds to set of textual diffs (i.e. UNIX diffs) grouped by file and function name for all the changes between two successive versions of a program for revisions v_i and v_{i+1}. Algorithm 5 iterates over the set of revision numbers V and processes both interprocedural and intraprocedural updates. New functions added in revision v_{i+1} are added to $\mathcal{MV}$ on lines 8-12. The **GetNewFunctions** procedure retrieves a list of function names that were generated in the preprocessing phase. These function names are used to retrieve the CFGs generated in the preprocessing phase for the new functions and added to $\mathcal{MV}$.

The intraprocedural updates for the changed files between versions v_i and v_{i+1} are processed on lines 13-16. The set changed flles are retrieved from the preprocessing phase to process diffs grouped by file name and function. Updates are made for the group of diffs in Algorithm 6 which defines the **UpdateMVCFG** procedure on line 15 of Algorithm 5.

The file name of a modified file is used in Algorithm 6 to retrieve a list of function names

```
   Input  : FileName, $v_{i+1}$, $\Gamma$, $Q$
   Output: MVCFG with intraprocedural update related to unit
 1 Procedure UpdateMVCFG $(FileName, v_{i+1}, \Gamma, Q)$
 2 $DiffGroups \leftarrow$ GetTextualDiffs $(FileName, v_{i+1})$
 3 foreach $(FuncName, DiffList) \in DiffGroups$ do
 4     $(N_{v_{i+1}}, E_{v_{i+1}}, S, T, K_{v_{i+1}}, x) \leftarrow$ BuildCFG $(FuncName, v_{i+1})$
 5     $\Phi_{v_i} \leftarrow$ CreateNodeMapping $(FuncName, DiffList)$
 6     $(N, E, S, T, K, x, \widehat{\gamma}, \mu) \leftarrow \Gamma[x]$
 7     foreach $(p_a, l_a, p_d, l_d) \in DiffList$ do
 8         if $l_a > 0$ then
 9             if $l_d > 0$ then
10                 DeleteCFGDiff $(N, N_{v_{i+1}}, E, E_{v_{i+1}}, K, \Phi_{v_i}, \mu, v_i, v_{i+1}, p_d, l_d)$
11             end
12             InsertCFGDiff $(N, N_{v_{i+1}}, E, E_{v_{i+1}}, K_{v_{i+1}}, Q, \Phi_{v_i}, \mu, \widehat{\gamma}, v_i, v_{i+1}, p_a, l_a)$
13         end
14         else
15             DeleteCFGDiff $(N, N_{v_{i+1}}, E, E_{v_{i+1}}, K, \Phi_{v_i}, \mu, v_i, v_{i+1}, p_d, l_d)$
16         end
17     end
18     UpdateVersions $(S, E, \mu, v_{i+1}, \Phi_{v_i})$
19 end
20 UpdateLineNumbers $(FileName, \Gamma, v_{i+1}, \Phi_{v_i})$
21 End Procedure
```

Algorithm 6: The procedure updates the MVCFGs that represent the functions in a file using the UNIX style diffs for file revisions between program versions v_i and v_{i+1}.

of changed functions generated in the preprocessing phase. The MVCFGs that represent the changed functions in a file are updated using the UNIX style diffs from the code changes between program versions v_i and v_{i+1}. The UNIX style diffs are retrieved on line 2, and the diff groups are processed on lines 3-22 for each changed function. The preprocessed CFG is retrieved with the **BuildCFG** procedure on line 4. The node mapping is created using the line numbers of the diffs and the starting line number of the the function with the **CreateNodeMapping** procedure using the information from the preprocessing phase. The Γ function is used to lookup the MVCFG using the function signature x on line 6.

The loop on lines 7-20 updates an MVCFG with the textual diffs for changes in a function. Each diff has the UNIX diff format (p_a, l_a, p_d, l_d) where p_a is the starting line number for inserted lines of code, l_a is the number of lines inserted, p_d is the starting line number for deleted lines of code, and l_d is the number of lines deleted. If a diff does not contain any deleted lines, both $p_d = 0$ and $l_d = 0$. Likewise, both $p_a = 0$ and $l_a = 0$ when a diff contains no inserted lines.

UNIX style diffs have three forms. Each diff is either an insert, delete, or update. An insert type diff only adds new lines in a file change so both $p_d = 0$ and $l_d = 0$. On the other hand, a delete type diff only deletes lines in a file change so both $p_a = 0$ and $l_a = 0$. The third type of diff is an update that inserts and deletes lines in a file change. These three types of diffs are handled as separate cases, and each case is handled using the **InsertCFGDiff** and **DeleteCFGDiff** procedures defined in Algorithms 7 and 8 respectively.

Both Algorithms Algorithms 7 and 8 work together and perform similar tasks. The procedures defined by these algorithms do the work of translating textual diffs into the nodes and edges for the related graph diff used to update the MVCFG. These algorithms also label the edges with the correct version numbers and create the v-branch and v-merge nodes where control flow paths diverge or converge across versions.

The Algorithm 7 defines the procedure that is used in Algorithm 6 to update an MVCFG with a graph diff related to deleted lines of code. The **BuildCFGDiff** function on line 2 uses p_d and l_d to select the nodes and incident edges from N, E and K of the MVCFG. The graph diff $(\Delta N_{v_i}, \Delta E_{v_i}, \Delta K_{v_i})$ correspond to the nodes related to the program statements deleted and all edges incident to the selected nodes. The set of incident edges ΔE_{v_i} is processed on lines 3-24 to create or update any v-branch and v-merge nodes related to the deleted lines. The edges $(n_1, n_2) \in \Delta E_{v_i}$ that are incident with adjacent nodes such that $n_1, n_2 \in \Delta N_{v_i}$ do not need updating since the edges and nodes are contained in the deleted component of the control flow graph. Only the edges that are incident with a node in $N_{v_{i+1}}$ such that $n_1 \in_\Phi N_{v_{i+1}}$ or $n_2 \in_\Phi N_{v_{i+1}}$ need labels updated to mark points of convergence or divergence across versions.[5]

A v-branch node is created for a node that is incident to an edge such that $(n_1, n_2) \in$

[5] The $\in_\Phi$ operation is used whenever the node mapping function is required to determine when a node from v_i also belongs to v_{i+1}.

```
1  Procedure DeleteCFGDiff ($N, N_{v_{i+1}}, E, E_{v_{i+1}}, K, \Phi_{v_i}, \mu, v_i, v_{i+1}, p_d, l_d$)
2  $(\Delta N_{v_i}, \Delta E_{v_i}, \Delta K_{v_i}) \leftarrow$ BuildCFGDiff ($N, E, K, v_i, p_d, l_d$)
3  foreach $(n_1, n_2) \in \Delta E_{v_i}$ do
4      if $n_1 \in_\Phi N_{v_{i+1}}$ then
5          MakeVBranchNode ($n_1, E, v_i, \mu$)
6          $A_{out} \leftarrow$ AdjOut ($n_1, E_{v_{i+1}}, \Phi_{v_i}$)
7          foreach $a \in A_{out}$ do
8              if $a \notin_\Phi E$ then
9                  $E \leftarrow E \cup \{a\}$
10                 $\mu[a] \leftarrow < v_{i+1}, v_{i+1} >$
11             end
12         end
13     end
14     else if $n_2 \in_\Phi N_{v_{i+1}}$ then
15         MakeVMergeNode ($n_2, E, v_i, \mu$)
16         $A_{in} \leftarrow$ AdjIn ($n_2, E_{v_{i+1}}, \Phi_{v_i}$)
17         foreach $a \in A_{in}$ do
18             if $a \notin_\Phi E$ then
19                 $E \leftarrow E \cup \{a\}$
20                 $\mu[a] \leftarrow < v_{i+1}, v_{i+1} >$
21             end
22         end
23     end
24 end
25 End Procedure
```

Algorithm 7: The procedure uses a UNIX diff to update an MVCFG with a graph diff related to deleted lines of code in v_i.

ΔE_{v_i} and $n_1 \in_\Phi N_{v_{i+1}}$. The v-branch nodes are created or updated for n_1 on lines 4-13. The **MakeVBranchNode** procedure uses the lazy labelling method to update the edges $(n_1, m_2) \in E$ that are incident with the node n_1 (see Appendix A for a definition of **MakeVBranchNode**). The incident edges are labelled for version v_i when **MakeVBranchNode** is finished updating μ. The next lines 6-12 updates the labels for any incident edges that new in v_{i+1}. The **AdjOut** procedure is used to retrieve all the edges in $E_{v_{i+1}}$ that are incident with and directed out from n_1. Any new edges a such that $a \notin_\Phi E$ are added to E and labelled with $< v_{i+1}, v_{i+1} >$.

If $(n_1, n_2) \in \Delta E_{v_i}$ and $n_2 \in_\Phi N_{v_{i+1}}$, then a v-merge node is created or updated for

```
Procedure InsertCFGDiff ($N, N_{v_{i+1}}, E, E_{v_{i+1}}, K_{v_{i+1}}, Q, \Phi_{v_i}, \mu, \widehat{\gamma}, v_i, v_{i+1}, p_a, l_a$)
$\Delta C \leftarrow$ BuildCFGDiff ($N_{v_{i+1}}, E_{v_{i+1}}, K_{v_{i+1}}, v_{i+1}, p_a, l_a$)
$(\Delta N_{v_{i+1}}, \Delta E_{v_{i+1}}, \Delta K_{v_{i+1}}) \leftarrow \Delta C$
foreach $(n_1, n_2) \in \Delta E_{v_{i+1}}$ do
    $a \leftarrow (n_1, n_2)$
    if $n_1 \in_{\Phi} N$ then
        MakeVBranchNode ($n_1, E, v_i, \mu$)
        $\mu[a] \leftarrow < v_{i+1}, v_{i+1} >$
    end
    else if $n_2 \in_{\Phi} N$ then
        MakeVMergeNode ($n_2, E, v_i, \mu$)
        $\mu[a] \leftarrow < v_{i+1}, v_{i+1} >$
    end
    else
        $\mu[a] \leftarrow \emptyset$
    end
    $E \leftarrow E \cup \{a\}$
end
$N \leftarrow N \cup \Delta N_{v_{i+1}}$
$K \leftarrow K \cup \Delta K_{v_{i+1}}$
Enqueue ($Q, \Delta K_{v_{i+1}}, \widehat{\gamma}$)
End Procedure
```

Algorithm 8: The procedure uses a UNIX style diff to update an MVCFG with a graph diff related to inserted lines of code in v_{i+1}.

n_2. V-merge nodes are created or updated for n_2 on lines 14-22. The lazy labelling method is used to update the edges $(m_1, n_2) \in E$ with the **MakeVMergeNode** procedure on line 15 (see Appendix A for a definition of **MakeVMergeNode**). The **MakeVMergeNode** procedure updates labels with the version number v_i. New edges incident with v-merge nodes are labelled and added to E on lines 16-21.

The Algorithm 8 defines the procedure that is used in Algorithm 6 to update an MVCFG with a graph diff related to inserted lines of code. The **InsertCFGDiff** procedure is used to process the two remaining types of UNIX style diffs. The procedure is used on line 12 of Algorithm 6 to add new nodes and edges from changes related to revision v_{i+1}. The algorithm is used alone to process insert type diffs related to new program statements. Otherwise, Algorithm 8 is used together with Algorithm 7 for update type diffs where program statements are both added and deleted with respect to the same line number.

The **BuildCFGDiff** function on line 2 uses p_a and l_a to select the new nodes and incident edges from $N_{v_{i+1}}, E_{v_{i+1}}$, and $K_{v_{i+1}}$. The nodes and edges for the graph diff $\Delta N_{v_{i+1}}, \Delta E_{v_{i+1}}$, and $\Delta K_{v_{i+1}}$ correspond to the new program statements inserted from program revision v_{i+1}. Each edge in $\Delta E_{v_{i+1}}$ is processed on lines 4-18 and added to the MVCFG on line 17. If any edge $(n_1, n_2) \in \Delta E_{v_{i+1}}$ is incident with a node that maps to a node from the previous version v_i such that $n_1 \in_\Phi N$ or $n_2 \in_\Phi N$, then a v-branch node or v-merge node must be updated or created. Any new edge that is incident with a v-branch or v-merge node is labelled explicitly with $< v_{i+1}, v_{i+1} >$ on line 8 or line 12. Otherwise, the edge is labelled with $\emptyset$. Once all edges are labelled and added to the MVCFG, the new nodes are added on lines 19 and 20. The new call nodes are placed in a queue Q to be updated when all the CFG diffs have been processed for v_{i+1}.

Algorithms 7 and 8 have some overlap of incident edges that are labelled for v-branch and v-merge nodes. It is possible for two different diffs to have incident edges that share common v-merge or v-branch nodes. However, the overlap does not create an incorrect MVCFG since new edges incident to v-merge or v-branch nodes are always explicitly labelled with $< v_{i+1}, v_{i+1} >$. The **MakeVBranchNode** and **MakeVMergeNode** only replace the implicit label $\emptyset$. The node mapping Φ and maintaining the correct line number labels for nodes becomes important when incrementally updating the MVCFG edge set E and the edge label map μ.

After Algorithm 6 exhausts a list of diffs for a function update on lines 7-17, the remaining labels from previous versions must be updated. Some edges common to E_i and E_{i+1} that are not selected in diffs that were processed might have explicit labels of the form $< v_j, v_i >$ where $j <= i$. These labels must be updated with the correct label $< v_j, v_{i+1} >$. The procedure **UpdateVersions** is used to update the remaining labels. This procedure uses a breadth first search beginning with the start node S and searches the graph for v_{i+1} to find the remaining edges. An impasse is used as the method for selecting labels to update. An impasse occurs when a breadth search visits a v-branch node or a node incident to v-merge node that does not have an edge directed out with a label for v_{i+1}. All the incident edges directed out from an impasse node with the label $< v_j, v_i >$ are updated to $< v_j, v_{i+1} >$.

After all the diffs for file changes have been processed on lines 3-19 in Algorithm 6, the versioned line numbers used to label nodes must be updated to reflect the correct node line numbers for v_{i+1}. The **UpdateVersions** procedure is responsible for updating line numbers for nodes in common to both v_i and v_{i+1} using the node mapping function Φ. The functions that did not have any diffs might be impacted by line number shifts caused by changes in other functions sharing the same file. The MVCFGs corresponding to the unchanged functions must be updated to preserve the integrity of the node mapping functions Φ. The node line numbers for the MVCFGs representing the unchanged functions are updated on line 20 using the **UpdateLineNumbers** procedure. This procedure calculates the line number offsets created by program changes in other functions and updates the node labels for each MVCFG with the new line numbers and revision number v_{i+1}.

When all the file names are processed in Algorithm 5 on lines 14-16, the changes for the MVCFGs are finalized for revision v_{i+1} by updating the call node mapping functions $\widehat{\gamma}$ with the new call nodes. This procedure follows the same steps on lines 14-19 in Algorithm 1. The MVICFG is complete when all revision numbers are exhausted in V.

Chapter 4

Experimental Results

The goal of the experiments is to demonstrate that the modified algorithm for constructing the MVICFG gererates a correct representation of a sequence of interprocedural control flow graphs for a revision history of moderately sized programs. The efficiency and scalability of the modified algorithm to construct the MVICFG is also demonstrated. The experiments will also show that the MVICFG has practical applications for software verification of changes to help combat the effects of software aging.

4.1 Experimental Design and Implementation

The Hydrogen framework is composed of three major subsystems – preprocessor, MVICFG module, and Marple. The preprocessor is a stand alone system composed of scripts and tools that mine software repositories for information used by the MVICFG module and Marple. The MVICFG module is a plug-in for the MS Phoenix framework which processes the information from the preprocessing phase and builds on MVICFG representation for a benchmark in an experiment. The Marple tool analyzes the MVICFG to perform patch verification. Both the MVICFG module and Marple are implemented using the June 2008 edition of Microsoft Phoenix.

The preprocessor uses a myriad of scripts and technologies to mine a software repository. Python scripts are used to implement the Index Files stage and List New/Deleted Functions stage of the preprocessor. The Normalize Code stage is implemented with batch scripts to run the C/C++ preprocessor from Visual Studio 8. The Parse Functions stage uses srcML and MS XSL to extract functions. The Generate Diffs stage is implemented using the UNIX diff tool and batch scripts. The Build ICFGs stage generates an ICFG for each program version and stores each ICFG as an XML file using the June 2008 edition of Microsoft Phoenix.

There two sets of benchmarks from popular open source programs that are used to collect data for the experimental results. The first set of 4 benchmarks used in the first experiment contain multiple revisions and releases for the programs *gzip*, *flex*, *libpng*, and *ffmpeg*. The second set 7 benchmarks are used in the second and third experiments contain multiple releases of the programs *tcas*, *schedule*, *printtokens*, *gzip*, *tightvnc*, *libpng*, and *putty*.

The experiment in section 4.2.1 supports the hypothesis that the amount of code change in source code repositories is small, and the program versions in a revision history share a large percentage of common code. The second experiment in section 4.2.2 supports the hypothesis that the Hydrogen framework and the MVICFG will scale in time and memory for a large number of revisions in a software revision history. The first experiment measures the frequency of software changes to determine how well the MVICFG would scale for a sequence of revisions from a typical revision history. The second experiment collects data for the actual time and memory required by the Hydrogen framework to construct an MVICFG using the modified algorithm.

The third experiment in section 4.2.3 demonstrates that path sensitive analysis can be used to analyze the MVICFG to verify correctness of changes and address the problem of software aging. A sample of known bugs and respective patches are selected from the Common Vulnerability Exposure (CVE), Bugzilla, and other bug fixes discovered in source code repositories. Data is collected to evaluate how well Hydrogen verifies a bug fix for multiple releases and how accurately the releases impacted by the bug are detected.

4.2 Experimental Results

4.2.1 Code Change in Source Code Repositories

Table 4.1 shows the data collected for code change in minor releases and revisions for the benchmark programs *gzip*, *flex*, *libpng*, and *ffmpeg*. The two sections of the table titled *Revision* and *Release* show the results for two different samples of code change from the 4 benchmarks. The data samples for code revisions under the heading *Revision* are the results of a random sample of 50 single revisions from the source code repository (i.e. commits) for each benchmark. The data samples under the *Release* heading show the amount of code change for larger changes between minor releases of the software.

The random samples under the *Revision* section are selected by dumping a complete list of commit numbers for all the revisions in a source code repository. A random number generator is used to select a sample of 50 revisions to measure the amount of code change in a typical revision. The code change is measured by extracting the code from function definitions and using a UNIX diff utility to measure the lines of code change in a revision. The average numbers of modified and deleted functions for a revision are listed in the *M* and *D* columns respectively. The amount of interprocedural change is also measured in the A_I and A_M columns. The A_I column lists the average number of new functions in a revision that call other functions. The A_M column lists the average number of functions added that do not call other functions. Finally, the *S* column shows the percentage of code reuse in a revision. The percentage of Lines of Code (LOC) reuse for a revision between two program versions R_i and R_{i+1} is calculated by $S = \frac{TLOC_i - DLOC_i}{TLOC_i}$ where $TLOC_i$ is the total lines of code in R_i and $DLOC_i$ is the number of lines of code deleted in the revision of R_i.

The samples under the *Release* heading are not selected randomly, but measure the code change for all of the major and minor releases in a benchmark. The sample sizes vary for each benchmark since the number of releases is unique. All of the columns are measured the same as the samples of revisions. The main difference is that the change in releases

Table 4.1: Sample of Software Changes

Benchmark	Revision						Release					
	No.	**M**	**D**	A_I	A_M	**S**	**No.**	**M**	**D**	A_I	A_M	**S**
gzip	50	0.91	0.18	0.18	0	99.5%	5	3.6	3	0.4	0	98.0%
flex	50	1	0.14	0	0.86	97.8%	28	6.75	0.57	0.71	3.57	98.0%
libpng	50	2.65	0.52	0.2	27.8	98.8%	61	11.1	2.52	0.70	10.0	98.0%
ffmpeg	50	2	0.75	0	1.33	98.0%	6	3.67	0	0	0	99.99%

involve large groups of revisions.

The results in Table 4.1 shows that code changes impact a small number of functions and a small percentage of the code in program versions. The amount of shared code between successive program versions is an average of 98.5%. This means that the intersection of code in a sequence of program versions is high, and the memory or storage demands of the MVICFG would be significantly less than required for a sequence of ICFGs. The number of functions impacted by code changes also tends to be small. Thus, a path sensitive analysis such as demand driven analysis would perform better than other techniques that use traditional software assurance tools for verifying changes [6, 4].

4.2.2 Scalability of the MVICFG

The system used to build the MVICFG for the benchmarks in Table 4.2 has dual Intel Xeon E5520 CPU processors running at 2.27 GHz with 12.0 GB of RAM. The system used Windows 7 Enterprise OS 64-bit edition. The performance data is collected for the execution of the modified algorithm and does not include the memory size and execution times for the preprocessing phase or the demand driven analysis. [1]

Table 4.2 summarizes the results for the benchmark tests of the performance of the modified algorithm for constructing an MVICFG. The first column list the names of the open source benchmarks used. The *Versions* column is the number of program versions represented by the MVICFG. The *LOC* column is the number of lines of code in the first

[1] The results of this experiment were published in ICSE'14 paper related to this thesis [14].

Table 4.2: Scalability of Building MVICFGs

Benchmark	**Versions**	**LOC**	**Churn**	**ICFGs**	**MVICFG**	**T(s)**	**M (MB)**
tcas	40	173	6	6.6 k	476	9.5	59.7
schedule	9	412	6	2.4 k	298	5.1	58.5
printtokens	7	563	3.7	2.7 k	413	2.5	60.4
gzip	5	5.0 k	242	6.8 k	2.1 k	28.3	83.4
tightvnc	5	6.3 k	457.3	10.3 k	2.4 k	24.3	129.2
libpng	9	9.2 k	1.4 k	35.3 k	8.4 k	183.2	167.3
putty	5	34.5 k	8.2 k	28.3 k	13.3 k	2446.3	355.6

version of the benchmark. The *Churn* column gives the average lines of code per version for the lines of code impacted by a source code change between revisions. The *ICFGs* column is the total sum of nodes for each $ICFG_i$ where i is some version of the benchmark and $ICFG_i$ is an Interprocedural Control Flow Graph (ICFG) of a benchmark in the graph union of the MVICFG. The *MVICFG* column is the sum of nodes in the MVICFG representation of a benchmark. *T(s)* is the number of seconds it took to build the MVICFG for a benchmark. *M (MB)* is the number of megabytes (MB) consumed in the peak memory usage of the MVICFG build process.

The results show that the modified algorithm will scale for moderate sized open source C/C++ programs with thousands of lines of code. For example, it takes about 1.5 minutes to build an MVICFG for the *libpng* benchmark representing 9 releases of the program. The results also show that the algorithm scales for a benchmark with a relatively high churn. The results also show that the MVICFG significantly reduces the number nodes needed to represent all the releases of *libpng*. The one benchmark that did not perform well was the *putty* benchmark. Some factors contributed to the degraded performance. The *putty* benchmark had the highest amount of churn than any of the other benchmarks. Consequently, a high number of v-branch and v-merge nodes are created per function change in this benchmark. Performance analysis of the algorithm implementation showed that the a loss of performance was due to a high number of file read operations. Since the modified algorithm scans nodes selected by each diff for incident edges in the ICFG files generated in the preprocessing phase, a high number of file read operations were a result. Thus, the results show that the modified algorithm performs best when the amount of churn is small between program versions.

The correctness of the MVICFG is validated by selecting parts of the MVICFG and verifying correctness by inspection. The MVCFGs used in the patch verification in the experiments in section 4.2.3 are verified correct by dumping the graphs to a dot graph and a text representation. The debug information that MS Phoenix provides is extended using two complementary dump formats (see Appendix C for more details).

4.2.3 Bug Impact and Patch Verification

The Table 4.3 shows the results for applying demand driven path sensitive analysis to the MVICFG to verify bug patches for multiple releases of a program. The experiment is conducted by constructing an MVCFG for a release of a program that contains a known bug. The experiment is run on Windows 7 system with a duo core i7-2600 CPU and 16 GB of RAM.[2]

The name and release number of each benchmark is listed in the *Benchmark* column of the Table 4.3. The different types of bugs detected in the demand driven analysis are listed in the *Bug* column. The different types of bugs that are detected are denoted by the following (1) *BO* is a buffer overrun, (2) *IO* is an integer overflow, (3) *IS* is an integer signedness bug, and (4) *NP* is a null pointer dereference. In the *Fixed* column, the result *Yes* indicates that Hydrogen determines the patch fixes the bug, and *No* indicates that Hydrogen determines the patch fails to fix the bug. Under the *Incremental Analysis* heading, the first *T(s)* column shows the amount of time needed to detect the bug, and the second *T(s)* column is the time taken to verify the bug fix.

The columns under the *Multiversion Analysis* heading shows the results of the analysis for multiple releases of a program. The *Releases* column shows the total number of releases that are analyzed including the buggy release. The *Doc* and *Impacted* columns show how

[2] The results of this experiment are published in the ICSE'14 paper related to this thesis [14].

Table 4.3: Determining Bug Impact and Verifying Fixes

Benchmark	**Incremental Analysis**				**Multiversion Analysis**					
	Bug	**T(s)**	**Fixed**	**T(s)**	**Releases**	**Doc**	**Impact**	**T(s)**	**Fixed**	**T(s)**
gzip-1.2.4	BO	0.12	Yes	0.08	4	1	4	0.13	4	0.69
libpng-1.5.14	IO	0.28	No	0.24	6	2	6	0.48	0	1.45
libpng-1.5.14	IO	0.24	Yes	0.18	6	2	6	1.45	5	1.22
tightvnc-1.3.9	IS	2.6	Yes	0	4	1	4	4.8	4	0
putty-0.55	NP	20.0	Yes	0.07	3	1	1	26.1	1	0.09

the documented number of releases impacted by the bug compares with the number of impacted releases detected by the Hydrogen framework. The *Fixed* column shows the number of releases that Hydrogen determines the patch successfully fixes. The *T(s)* columns show the amount of time Hydrogen takes to detect the number of buggy releases and the amount of time to verify the patch for the releases.

The *Incremental Analysis* demonstrates the speed with which Hydrogen can verify a patch between a buggy release and a patched version of the release. The patch verification is performed quickly for all types of bugs analyzed in this study. The patches for *libpng-1.5.14* on the 2^{nd} and 3^{rd} rows show an example of a failed attempt to fix a bug. The first patch in the second row was documented as a correct patch even though it was later discovered to be an incorrect patch. The example shows that Hydrogen successfully verified that the first patch was incorrect while the second patch was correct.

The results under the *Multiversion Analysis* shows that Hydrogen was able to detect more releases impacted by a bug than the documentation reported. A code inspection revealed that Hydrogen correctly identifies the impacted releases that were not reported in the documentation. Hydrogen fails to verify the patch on the 3^{rd} row for all the impacted releases for *libpng*.

Chapter 5

Conclusions

5.1 Related Work

Several techniques are currently available for use in multi-version program analysis. Techniques range in application such as software version merging, program differencing, change classification, and regression testing [12]. The Hydrogen framework combines program differencing techniques and demand driven analysis for bug detection and verifying bug fixes in software changes.

Current approaches for differencing program versions use either a lexical, syntactical, or semantic differencing approach [10, 7, 5, 17, 1, 2, 11, 13, 9]. A well known differencing technique for displaying lexical differences between files is the UNIX diff tool [10]. The UNIX diff tool uses a line-by-line comparison based on Hirshberg's longest common subsequence algorithm to detect textual differences between files. The UNIX diff cannot distinguish between changes in whitespace, comment changes, or changes in code. The modified algorithm in the Hydrogen framework leverages the differential information from UNIX diffs by translating UNIX diffs into interprocedural control flow changes.

Another differencing approach called ChangeDistiller makes use of the abstract syntax tree (AST) of a program with tree differencing techniques to detect syntactic and structural changes in programs [7, 5]. ChangeDistiller is used in a software evolution platform called Evolizer to analyze and classify source code changes [8]. A similar approach called Dex also uses a tree differencing technique for analyzing syntactic and semantic changes [17]. The MVICFG in the Hydrogen framework is used to detect behavioral changes in a program related to control flow and data flow changes rather than structural changes in code. The Hydrogen framework is complementary to ChangeDistiller and Evolizer or Dex by extending the work for classifying bug introducing and bug fix changes.

The JDiff technique uses enhanced control flow graphs and hammocks to analyze semantic change in object oriented programs [1, 2]. The technique is based on graph isomorphisms and uses a construct called hammocks to detect both interprocedural and intraprocedural changes in methods in an object oriented program. The Hydrogen framework does not address changes specific to object oriented programs, but the MVICFG can be used to detect interprocedural changes in object oriented programs that effect call flow changes. Also, the framework in this study is not limited to graphical differences between two versions of a program, but can be used to analyze changes across multiple versions of a program.

Other differencing approaches can be used to detect semantic (behavioral) differences between program versions. One tool called SemanticDiff is used to compare observable input-output behavior[11]. A newer tool called SymDiff uses an intermediate verification language called Boogie and symbolic analysis to approximate how syntactic changes impact runtime behavior without running the program versions [13]. Hydrogen uses static analysis to translate textual changes between multiple versions of a program into control flow and data flow changes. The changes are analyzed using demand-driven analysis to detect behavioral changes associated with bug detection and bug fix verification [3].

In the BUGINNINGS article [18], the authors describe a new approach to tracking and identifying bug-introducing changes to code. The authors use program dependence graphs to define bug regions in code using control dependences, data dependences, and call relations. The Hydrogen framework uses symbolic analysis to detect change of invariance. Symbolic analysis would also replace the text based approach in BUGINNINGS for identifying semantics changes not identified by program dependences. Hydrogen not only can detect bug origins, but also uses demand driven analysis for detecting types of defects such as null pointers, buffer overflows, integer overflows, etc. to verify removal in consecutive versions.

5.2 Future Work

The MVICFG is a general data structure that can used to analyze control flow change across a sequence of interprocedural control flow graphs. New applications for the MVICFG need to be explored to demonstrate the usefulness of the Hydrogen framework. One area of investigation would explore patterns in the progression of changes in program properties across multiple versions of a program. Patterns of invariant change or patterns of buggy changes might be used to identify incorrect changes in a revision history. The MVICFG would provide an effective framework for caching intermediate analysis results related to commonalities across multiple versions of a program that would be used improve the efficiency of program analysis.

More work is needed to show that the MVICFG can scale to larger benchmarks with a higher number of revisions than demonstrated by this study. Exploring ways of improving the algorithm for constructing the MVICFG might improve efficiency. Some problems with the implementation were not discussed in this thesis. In particular, the challenges of using line numbers to translate lexical diffs from the source code to graph diffs in the intermediate representation could have negatively impacted the performance of the implementation. There might also be some opportunities to improve the efficiency of the computational efficiency of the modified algorithm. Some experiments would help determine whether the trade-offs of the lazy labelling method with other edge labelling methods are worthwhile. The gains in memory efficiency from the lazy labelling method might not be worth the loss of computational efficiency that would be gained from other labelling methods.

The use of other diff techniques to create node maps might also improve the efficiency of the algorithm to construct the MVICFG. Other differencing techniques mentioned in the related works section might be adapted for use in the construction of the MVICFG. For instance, the tree differencing technique used in ChangeDistiller guided by lexical diffs mined from source code repositories might be a more efficient method for creating node maps [7, 5]. The graph differencing technique used in JDiff might also be used to create an MVICFG without the use node maps [1, 2]. New algorithm definitions could be created that would use alternative techniques for constructing the MVICFG. A comparison of these different approaches would help develop better algorithms.

The precision of the MVICFG could also be improved. For example, the MVICFG will often over represent control flow change by adding more nodes and edges than necessary to accurately represent a sequence of control flow graphs. The MVICFG is constructed by only comparing pairs of successive program versions using UNIX diffs without comparison to other program versions. For instance, program statements might be deleted in versions 1 and 2 of a program that were added in program version 3 as part of a revert commit. The current approach would add redundant nodes and edges in version 3 instead of labelling the existing edges as part of a reversion.

An important contribution of this thesis are the formal definitions of the MVICFG. The results of this thesis demonstrate the correctness of the MVICFG and the modified algorithm for constructing the MVICFG. However, more evidence is needed to show that the MVICFG and construction algorithm is correct for any sequence of program versions. Formal proofs of correctness are needed to prove that the MVICFG is a correct representation of any sequence of graphs. Proofs are also needed to show that the algorithms are correct.

5.3 Final Remarks

The MVICFG that represents multiple versions of a program is constructed by the Hydrogen algorithm using the differential information mined from a source code repository. The graph representation is analysed using techniques in demand driven analysis to detect bugs and verify bug patches in software changes. The Hydrogen algorithm provides a well defined method for constructing a graph representation of multiple versions of a program from lexical differential information, and the experimental results show that this method is efficient and correct. A worst case analysis for storage space required for interprocedural control flow graphs used in multi-version program analysis provides a rule determining how efficiently the MVICFG stores a sequence of graphs. A theoretical reduction is possible for revision histories with small incremental changes, and experimental evidence from the benchmarks verifies that a reduction in storage space can be achieved. An experimental analysis also demonstrates the effectiveness and efficiency of bug detection and bug fix verification using the MVICFG.

The most important contributions of this thesis are the formal definitions of the MVICFG and the detailed definitions of the algorithms used to construct the MVICFG. The definitions of this thesis are the foundation of the MVICFG and the blueprints for implementing the algorithms needed to construct an MVICFG. The definitions can be used as a starting point for developing better algorithms for constructing the MVICFG as well as provide the tools needed for general proofs of correctness.

Bibliography

[1] T. Apiwattanapong, A. Orso, and M.J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. of 19th IEEE Inter. Conf. on Automated Software Engineering*, pages 2–13, September 2004.

[2] T. Apiwattanapong, A. Orso, and M.J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. In *Automated Software Engineering*, volume 14, pages 3–36, March 2007.

[3] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Proc. of 6th European Software Engineering Conf. held jointly with 5th ACM SIGSOFT Inter. Symp. on Foundations of Software Engineering*, pages 361–377, 1997.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic program errors. In *Software-Practice and Experience.*, pages 775–802, 2000.

[5] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proc. of ACM SIGMOD Inter. Conf. on Management of Data*, pages 493–504, 1996.

[6] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proc. of ACM SIGPLAN 2002 Conf. on Programming Languages and Design Implementation*, pages 57–68, 2002.

[7] B Fluri, M Wursch, M Pinzger, and H C Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. In *IEEE Trans. on Software Engineering*, volume 33, pages 725–743, 2007.

[8] H C Gall, B Fluri, and M Pinzger. Change Analysis with Evolizer and ChangeDistiller. In *IEEE Software*, volume 26, pages 26–33, 2009.

[9] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. In *ACM Trans. on Programming Language and Systems*, volume 11, pages 345–387, 1989.

[10] James W. Hunt and M. Douglas McIlroy. An Algorithm for Differential File Comparison. Technical report, Bell Laboratories, 1976.

[11] D Jackson and D A Ladd. Semantic Diff: a tool for summarizing the effects of modifications. In *Proc. on Inter. Conf. on Software Maintenance*, pages 243–252, 1994.

[12] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proc. of Inter. Workshop on Mining Software Repositories*, pages 58–64, 2006.

[13] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: a language-agnostic semantic diff tool for imperative programs. In *Proc. of 24th Inter. Conf. on Computer Aided Verification*, pages 712–717, 2012.

[14] Wei Le and Shannon D Pattison. Patch Verification via Multiversion Interprocedural Control Flow Graphs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1047–1058, 2014.

[15] T Mens, M Wermelinger, S Ducasse, S Demeyer, R Hirschfeld, and M Jazayeri. Challenges in software evolution. In *Proc. of 8th Inter. Workshop on Principles of Software Evolution*, pages 13–22, 2005.

[16] David Lorge Parnas. Software aging. In *Proc. of 16th Inter. Conf. on Software Engineering*, pages 279–287, 1994.

[17] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. In *Proc. of 20th IEEE Inter. Conf. on Software Maintenance*, pages 188–197, September 2004.

[18] Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao. BUGINNINGS: identifying the origins of a bug. In *Proc. of 3rd India Software Engineering Conf.*, pages 3–12, 2010.

[19] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proc. of ACM SIGSOFT 20th Inter. Symp. on the Foundations of Software Engineering*, pages 1–11, 2012.

[20] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 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. ACM, 2011.

Appendix A

Supporting Algorithms

A.1 Support for the General Algorithm

Algorithms 9 and 10 define procedures that are used by Algorithm 1 and 2.

```
Procedure LabelEdges ($E, i$)
foreach $a \in E$ do
    $\alpha[a] \leftarrow \{i\}$
end
return $\alpha$
End Procedure
```

Algorithm 9: Algorithm used to label edges

```
Procedure CompareProc ($\mathcal{R}_i, \mathcal{R}_{i+1}$)
$\mathcal{R}' \leftarrow \emptyset$
$\mathcal{R}^+ \leftarrow \mathcal{R}_{i+1}$
foreach $(x, P_i) \in \mathcal{R}_i$ do
    if SymbolExists ($\mathcal{R}_{i+1}, x$) then
        $P_{i+1} \leftarrow$ GetProcedure ($\mathcal{R}_{i+1}, x$)
        $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(x, P_i, P_{i+1})\}$
        $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - (x, P_{i+1})$
    end
end
return $(\mathcal{R}', \mathcal{R}^+)$
End Procedure
```

Algorithm 10: Algorithm used to compare procedures of two versions of a program

A.2 Support for the Modified Algorithm

Algorithms 11 and 12 define procedures that are used by Algorithms 7 and 8.

```
Input  : $n$,$E$,$v$,$\mu$
Output: Creates a v-branch node.
Procedure MakeVBranchNode ($n$,$E$,$v$,$\mu$)
$A_{out} \leftarrow$ AdjOut ($n, E$)
foreach $a \in A_{out}$ do
    $T \leftarrow \mu[a]$
    if $T = \emptyset$ then
        $v_{i_1} \leftarrow$ FindMinVersion ($n, E, \mu$)
        $\mu[a] \leftarrow < v_{i_1}, v >$
    end
end
End Procedure
```

Algorithm 11: Creates a v-branch in the modified algorithm for building an MVICFG using UNIX diff information.

```
Input  : $n$,$E$,$v$,$\mu$
Output: Creates a v-merge node.
Procedure MakeVMergeNode ($n$,$E$,$v$,$\mu$)
$A_{in} \leftarrow$ AdjIn ($n, E$)
foreach $a \in A_{in}$ do
    $T \leftarrow \mu[a]$
    if $T = \emptyset$ then
        $(m, n) \leftarrow a$
        $v_{i_1} \leftarrow$ FindMinVersion ($m, E, \mu$)
        $\mu[a] \leftarrow < v_{i_1}, v >$
    end
end
End Procedure
```

Algorithm 12: Creates a v-merge in the modified algorithm for building an MVICFG using UNIX diff information.

Appendix B

Libpng Case Study

B.1 Overview

An important outcome of this thesis is the demonstration of the usefulness of the Hydrogen framework. The case study for the Libpng benchmark was conducted to find interesting patches that could be use to demonstrate the usefulness of the Hydrogen framework. The objective of this case study was to discover opportunities for Hydrogen applications in patch verification for current and future studies. Two random samples of bug reports are examined, and some of the attributes for each corresponding code change in the source code repository are catalogued. The first sample in section B.2 looks at the types of defects that are fixed in a random sample of bug reports and the impact of the code changes used to fix the defect. The second sample in section B.3 was examined to discover how many defects impact multiple releases and what percentage of code patches had to be modified to apply to multiple releases. The patches that were selected as Libpng benchmarks for use in the results of this thesis come from this case study.

B.2 Candidate Patches for Static Verification

Total Bug Reports: 213

Total Closed Fixed Bugs: 125

Static Analyzable Bugs: 15%

The statistics above summarize the number of bugs found in the bug reporting system on SourceForge for the Libpng project. There are 213 tickets in the bug tracker with 16 open tickets and 197 closed tickets. Only the closed tickets are interesting for change verification since we are interested in statically analyzing software revisions related to bug fixes. However, not all closed tickets are interesting because some closed tickets are not associated with changes. Some research is required to get a good approximation of closed tickets that are related to actual changes in the Libpng repository.

There are several different statuses for closed tickets in the Libpng bug reporting system. The statuses include closed, closed-accepted, closed-out-of-date, closed-invalid, closed-later, closed-rejected, closed-works-for-me, closed-duplicate, closed-wont-fix, and closed-fixed. The statuses that are clearly not related to changes in the repository are closed-invalid, closed-rejected, closed-works-for-me, closed-duplicate, and closed-wont-fix. The statuses closed, closed-accepted, closed-out-of-date, and closed-later appear to be used infrequently and are not always associated with changes in the repository. The only status that is clearly related to changes in the repository is closed-fixed. There are 125 tickets with the closed-fixed status.

Random sampling was used to estimate the percent of closed fixed bugs that are static analyzable. The reports for the 125 closed-fixed bugs were exported from the bug reporting system as an RSS document. A sample of 20 bug reports were selected from the RSS document using a random sequence generator. There were three static analyzable bugs found in the sample: integer overflow, buffer overflow, and null pointer. Thus, 15% of bugs in the reporting system appear to be static analyzable.

The random sample of bug reports taken in the previous section has information about patches that impact multiple releases. Table B.1 summarizes the bug report sample to show which types of bugs impact multiple releases. The table shows 45% of bugs sampled had patches that impacted multiple releases. All of the static analyzable bugs in the sample have patches that impact multiple releases. The sample would suggest that static analyzable bugs are more likely to effect multiple releases than other types of bugs. Further study would be required to assess whether or not patches for buffer overruns, integer overflows, and

Table B.1: Bug Report Sample for Libpng Benchmark

Bug ID	**Defect Type**	**Impacted Releases**	**Num of Releases**	**Static Analyzable**
203	Preprocessor	1.5.x, 1.6.x	2	False
199	Integer Overflow	1.4.x, 1.5.x	2	True
188	Specification	1.5.x, 1.6.x	2	False
179	Build	1.0.x, 1.2.x	2	False
158	Build	1.4.x	1	False
154	Text	1.4.x, 1.5.x	2	False
145	Preprocessor	1.2.x, 1.4.x	2	False
140	Preprocessor	1.2.x	1	False
133	Buffer Overrun	1.2.x, 1.4.x	2	True
129	Build	1.2.x	1	False
101	Preprocessor	1.2.x	1	False
93	Specification	1.2.x	1	False
89	Preprocessor	1.2.x	1	False
85	Build	1.2.x	1	False
81	Null Pointer	1.2.x, 1.4.x	2	True
78	Preprocessor	1.2.x, 1.4.x	2	False
76	Preprocessor	1.2.x	1	False
55	Preprocessor	1.2.x	1	False
32	?	?	?	?

null pointers often impact multiple releases in projects that support multiple maintenance releases.

There are many other bugs of interest that get patched in the repository that do not appear in the bug tracking system for Libpng. Other bugs fixes can be found listed in the CHANGES document in the Libpng project folder for each release. No explicit reasons were found for not adding every bug to the tracking system, but there is clear evidence that not all Libpng bugs fixes are tracked.

An example of a buffer overrun bug not found in the bug tracker but listed in the CHANGES document was released February 27, 2012 in the 1.4.10beta01 release. The patch for this bug impacts three different releases. The patches for each release can be seen by clicking the links listed below.

- Patch 1.0.x
- Patch 1.2.48
- Patch 1.4.10

This patch was directly applicable to the three releases without any modifications to the patch. The code effected by tho patch was essentially unchanged since the png_handle_sCAL function was added in the 1.0.x release. The patch was not applicable to the first major release 0.x since the png_handle_sCAL function did not exist. The patch was not applicable to the releases after 1.4.x because the statements causing the buffer overflow were replaced by new functions added in successive releases.

An interesting feature of the bug report sample is the high frequency of bugs caused by incorrect preprocessor definitions. Table B.1 shows that 45% of bug reports are related to incorrect preprocessor definitions. The Libpng project relies heavily on the use of preprocessor defs and conditional compilation. Issues with preprocessor defs are not discovered unless a specific preprocessor definition is selected in the compilation. There appear to be a large number of possible combinations for these preprocessor definitions. Unfortunately, there is not much information about valid combinations for these preprocessor definitions.

Another challenge related to conditional compilation are patches that cut across preprocessor definitions. The buffer overflow example in the previous section is an example of a cross cutting patch. Only part of the patch is selected when the PNG_FIXED_POINT_SUPPORTED definition is set. This is evidence that some patches must be tested using multiple combinations of preprocessor definitions.

This case study shows that a significant number of statically analyzable bugs impact multiple releases. 15% of bug reports from a random sample are static analyzable, and all of the static analyzable bugs impact multiple releases. In general, 45% of all bugs in the sample impact multiple releases. Conditional compilation was also observed to be a source of bug reports. The data suggests that it is difficult to detect bugs related to incorrect preprocessor definitions or bugs obscured by an uncommon preprocessor definitions. There seems to be an opportunity for better bug detection by analyzing multiple combinations of preprocessor definitions.

B.3 Modified Patches

Sample Size: 20

Multiple Releases: 8

% Modified Patches: 37.5%

The table below gives a list of bug reports for bug fixes in Libpng. Table B.2 shows which bug patches impact multiple releases. The 'Patch Modified' column indicates 'True' when the patch had to be modified to be applied to other releases. The list below summarizes some observations about the patches that were applied to multiple releases.

- Bug 203 was straight forward and did not require modifications.
- Bug 199 was a difficult that required modifications between releases as well as mutiple attempts to correctly fix the bug.
- Bug 188 was a large and difficult patch that involved many files. The patch was modified between releases.
- Bug 179 is a little confusing. It appears that the reverse of the patch applied to 1.0.x is applied to 1.2.x.
- Bug 154 is a simple documentation bug.
- Bug 145 uses a common patch between releases.
- Bug 133 patch appears to be simple and applied the same in two releases.
- Bug 81 share common code at time of the patch.

- Bug 78 – could not find the patch applied to release 1.4.x.

The results show that 37.5% of the defects that impact multiple releases used a patch that required modification to apply the fix across all impacted releases. This could be an opportunity for future studies to show how the Hydrogen framework would be used to automate patch modification and verification across multiple buggy releases.

Table B.2: Bug Report Sample for Libpng Benchmark

Bug ID	**Defect Type**	**Impacted Releases**	**Patch Modified**
203	Preprocessor	1.5.x, 1.6.x,1.7.x	False
199	Integer Overflow	1.4.x, 1.5.x	True
188	Specification	1.5.x, 1.6.x	True
179	Build	1.0.x, 1.2.x	True
158	Build	1.4.x	False
154	Text	1.4.x, 1.5.x	False
145	Preprocessor	1.2.x, 1.4.x	False
140	Preprocessor	1.2.x	False
133	Buffer Overrun	1.2.x, 1.4.x	False
129	Build	1.2.x	False
101	Preprocessor	1.2.x	False
93	Specification	1.2.x	False
89	Preprocessor	1.2.x	False
85	Build	1.2.x	False
81	Null Pointer	1.2.x, 1.4.x	False
78	Preprocessor	1.2.x, 1.4.x	?
76	Preprocessor	1.2.x	False
55	Preprocessor	1.2.x	False
32	?	?	?

Appendix C

MVCFG Inspection Method

C.1 Inspection of an MVCFG

Two formats are used to manually inspect an MVCFG for correctness. One format is the dot used to generate a PNG image file for a MVCFG that represents a function. The dot format is not documented, but the second that extends the MS Phoenix dump format for textual representation of flow graphs is also used in inspections. This format shows more details than the dot format, and it is documented in the following sections. The two formats used together are the main tools for verifying correctness and debugging the implementation.

C.2 Extended Format

The MVCFG dump format extends the MS Phoenix dump format for textual representation of flow graphs. The new extended format adds two new elements to the textual representation - block aliases and edge version summarization. Block aliases are used to unambiguously identify blocks in the multi-version control flow instead of relying on the block numbers used in the MS Phoenix dump format. The edge version summarization provides a summary of successor edges for each block with a list of version numbers related to each edge. The next two subsections provide details for the extended format.

C.2.1 Block Alias Format

The block alias format is a descriptive naming convention for flow graph blocks using the function name, version number, textual diff unit number, and block number related to the flow graph. There are two different formats for the alias. One format is used for blocks that belong to the initial version of the flow graph, and another format is used for blocks built using diff information added to the flow graph in subsequent versions.

The initial version of a function unit is referred to as the base version, and the alias will identify a flow graph block using BASE in the alias name. The following format is used for flow graph blocks belonging to the base version.

```
[function name]_BASE_BLOCK[block number]
```

Every block alias is prefixed with the function name related to the flow graph. The block number is an auto generated id unique to every block in the base version. The following example shows an alias for a block in a flow graph for a function named _main with a block number 1 in the base version.

```
_main_BASE_BLOCK1
```

The alias name format changes in subsequent versions to include the version number and diff unit number to help determine when the block was added to the flow graph. The following format is used for blocks added in subsequent versions.

```
[function name]_V[version number]_DIFF[diff unit number]
_BLOCK[block number]
```

The following example shows an alias for a block in a flow graph for a function named _main that was added in version 2 from diff unit 1 with a block number 2.

```
_main_V2_DIFF1_BLOCK2
```

C.2.2 Edge Version Summary

The MS Phoenix dump format includes information about predecessor and successor blocks using block numbers, but block numbers become ambiguous in the context of a multi-version flow graph. The extended format adds an edge version summarization at the end of the textual representation for each block using block aliases to identify successor blocks. This summarization lists all successor blocks by alias and lists the version numbers for the successor block. An asterisk is used to identify a successor block that is included in the control flow for all versions. The following format is used for the version summary.

```
[EVS] Edge Version Summary
Successor([block alias]) Version(s) [version numbers]
Successor([block alias]) Version(s) [version numbers]
...
```

The following edge version summarization is an example for a block that has three successor blocks. The block _main_BASE_3 is a successor in only version 1. The block _main_BASE_4 is a successor for all versions (i.e. versions 1,2,and 3). The block _main_V2 _DIFF1_BLOCK2 is a successor in versions 2 and 3.

```
[EVS] Edge Version Summary
Successor(_main_BASE_3) Version(s) 1
Successor(_main_BASE_4) Version(s) *
Successor(_main_V2_DIFF1_BLOCK2) Version(s) 2,3
```

C.3 MVCFG Example

Three consecutive versions of a simple program are provided to show a complete example of the textual representation for the extended flow graph dump for a multi-version flow graph. The three basic types of textual differences that are included in this example are insert, update, and delete. Version 2 of the program inserts 8 new lines on line number 8. Version 3 updates one line on line number 8 and deletes lines 10 through 12.

Example 1: Simple Program (version 1)

```
1 #include <stdio.h>
2
```

```
void main(int argc, char* argv[]){

        int i = argc;
        int j = i+2;
        printf(”j:␣%d”, j);


}
```

Example 2: Simple Program (version 2)

```
#include <stdio.h>

void main(int argc, char* argv[]){

        int i = argc;
        int j = i+2;
        printf(”j:␣%d”, j);
        j = i%2 - 1;
        int k = -argc * 2;
        j = k / j;
        i = k | 1;
        j = i ^ 7;
        int l = k & j;
        int m = l >> 3;
        int n = m << 2;
        printf(”i:␣%d,␣j:␣%d,␣k:␣%d,␣l:␣%d,␣m:␣%d,␣n:␣%d”,
                        i,j,k,l,m,n);
}
```

Example 3: Simple Program (version 3)

```
#include <stdio.h>

void main(int argc, char* argv[]){

        int i = argc;
        int j = i+2;
        printf(”j:␣%d”, j);
        j = i%3 + 1;
        int k = −argc * 2;
        int l = k & j;
        int m = l >> 3;
        int n = m << 2;
        printf(”i:␣%d,␣j:␣%d,␣k:␣%d,␣l:␣%d,␣m:␣%d,␣n:␣%d”,
                i,j,k,l,m,n);
}
```

MVCFG Dump Example

```
==== _main_BASE_BLOCK1
==== BasicBlock 1 Predecessor() Successor(2) next 2 pre 1 post 8 iDom 1
df () $L1: (references=0)
   {*StaticTag}, {*NotAliasedTag}, {*UndefinedTag}<1> = START _main(T)
[EVS] Edge Version Summary
      Successor(_main_BASE_BLOCK2) Version(s) *
==== _main_BASE_BLOCK2
==== BasicBlock 2 Predecessor(1) Successor(3,4) previous 1 next 3 pre 2
post 7 i Dom 1 df ()
_main: (references=1)
    _argc<2>, tv278<3> = ENTERFUNCTION
                         ENTERBLOCK ScopeSymbol267
                         ENTERBLOCK ScopeSymbol269
                         ENTERBLOCK ScopeSymbol272
    _i<4>              = ASSIGN _argc<2>
    _j<5>              = ADD 2, _i<4>
    t276               = CONVERT &??_C@_05JDOGFHNB@j?3?5?$CFd?$AA@
    {*CallTag}         = CALL* &_printf, t276, _j<5>, {*CallTag}, $L11(EH)
```

```
[EVS] Edge Version Summary
     Successor(_main_BASE_BLOCK3) Version(s) 1
     Successor(_main_BASE_BLOCK4) Version(s) *
     Successor(_main_V2_DIFF1_BLOCK2) Version(s) 2,3
==== _main_BASE_BLOCK3
==== BasicBlock 3 Predecessor(2) Successor(6) previous 2 next
4 pre 3 post 4 iDo m 2 df ()
                      EXITBLOCK ScopeSymbol272
                      EXITBLOCK ScopeSymbol269
                      EXITBLOCK ScopeSymbol267
                      RETURN 0, $L12(T)

[EVS] Edge Version Summary
     Successor(_main_BASE_BLOCK6) Version(s) *
==== _main_BASE_BLOCK4
==== BasicBlock 4 Predecessor(2) Successor(5) previous 3 next 5
pre 7 post 6 iDo m 2 df ()
$L11: (references=1)
                      GOTO {*StaticTag}, $L5
[EVS] Edge Version Summary
     Successor(_main_BASE_BLOCK5) Version(s) *
==== _main_V2_DIFF1_BLOCK2
==== BasicBlock 2 Predecessor() Successor(4) previous 1 next 4
                      GOTO $L4
[EVS] Edge Version Summary
     Successor(_main_V2_DIFF1_BLOCK4) Version(s) 1,2
     Successor(_main_V3_DIFF1_BLOCK2) Version(s) 3
==== _main_BASE_BLOCK6
==== BasicBlock 6 Predecessor(3) Successor(7) previous 5 next 7
pre 4 post 3 iDo m 3 df ()
$L12: (references=1)
                      GOTO {*StaticTag}, $L3
[EVS] Edge Version Summary
     Successor(_main_BASE_BLOCK7) Version(s) *
==== _main_BASE_BLOCK5
==== BasicBlock 5 Predecessor(4) Successor() previous 4 next 6
pre 8 post 5 iDom  4 df ()
$L5: (references=1)
                      UNWIND
==== _main_V2_DIFF1_BLOCK4
==== BasicBlock 4 Predecessor(2) Successor(5) previous 2 next 5
$L4: (references=1)
   t268               = REMAINDER _i, 2
   t269               = SUBTRACT t268
   _j                 = ASSIGN t269
                      GOTO $L5
```

```
[EVS] Edge Version Summary
      Successor(_main_V2_DIFF1_BLOCK5) Version(s) 1,2
==== _main_V3_DIFF1_BLOCK2
==== BasicBlock 2 Predecessor() Successor() previous 1 next 3
   t268              = REMAINDER _i, 3
   t269              = ADD t268, 1
   _j                = ASSIGN t269
[EVS] Edge Version Summary
      Successor(_main_V2_DIFF1_BLOCK5) Version(s) 3
==== _main_BASE_BLOCK7
==== BasicBlock 7 Predecessor(6) Successor(8) previous 6
next 8 pre 5 post 2 iDom 6 df ()
$L3: (references=1)
                       EXITFUNCTION
[EVS] Edge Version Summary
      Successor(_main_BASE_BLOCK8) Version(s) *
==== _main_V2_DIFF1_BLOCK5
==== BasicBlock 5 Predecessor(4) Successor(6) previous 4 next 6
$L5: (references=1)
   t272              = NEGATE _argc
   t273              = MULTIPLY t272, 2
   _k                = ASSIGN t273
                       GOTO $L6
[EVS] Edge Version Summary
      Successor(_main_V2_DIFF1_BLOCK6) Version(s) 1,2
      Successor(_main_V2_DIFF1_BLOCK7) Version(s) 3
==== _main_BASE_BLOCK8
==== BasicBlock 8 Predecessor(7) Successor() previous 7
pre 6 post 1 iDom 7 df ()
$L2: (references=0)
                       END
==== _main_V2_DIFF1_BLOCK6
==== BasicBlock 6 Predecessor(5) Successor(7) previous 5 next 7
$L6: (references=1)
   t275              = DIVIDE _k, _j
   _j                = ASSIGN t275
   t276              = BITOR _k, 1
   _i                = ASSIGN t276
   t277              = BITXOR _i, 7
   _j                = ASSIGN t277
                       GOTO $L7
[EVS] Edge Version Summary
      Successor(_main_V2_DIFF1_BLOCK7) Version(s) 1,2
==== _main_V2_DIFF1_BLOCK7
==== BasicBlock 7 Predecessor(6) Successor() previous 6 next 3
$L7: (references=1)
```

```
    t278                  = BITAND _k, _j
    _l                    = ASSIGN t278
    t280                  = SHIFTRIGHT _l, 3
    _m                    = ASSIGN t280
    t282                  = SHIFTLEFT _m, 2
    _n                    = ASSIGN t282
    t285                  = CONVERT &$SG3839
    {*CallTag}            = CALL &_printf, &_printf, t285, _i,
_j, _k, _l, _m, _n, {*CallTag}, $L3(EH)
[EVS] Edge Version Summary
      Successor(_main_BASE_BLOCK3) Version(s) 2,3
```

Appendix D

Proofs

D.1 Proof for Algorithm 3

Definitions:

Given a Φ_i relation defined in the definitions of section 2.2, the $\in_\Phi$ operator is defined for node sets and edge sets as follows. Given the node sets N, N_i, and N_{i+1} where $N_i \subseteq N$, let n be a node such that $n \in N_i$. We say that $n \in_\Phi N$ *iff* $\exists m \in N_{i+1}$ *s.t.* $\Phi_i(nl(n)) = nl(m))$. Furthermore, given the edge sets E, E_i, and E_{i+1} where $E_i \subseteq E$, let $a = (n_1, n_2)$ be an edge such that $a \in E_i$. We say that $a \in_\Phi E$ *iff* $\exists a' \in E_{i+1}$ where $a' = (m_1, m_2)$ *s.t.* $\Phi_i(nl(n_1)) = nl(m_1))$ and $\Phi_i(nl(n_2)) = nl(m_2))$.

Theorem

Given an edge $a \in E_{i+1}$ and $a \in_\Phi E$, if $\mu[a] \neq \emptyset$, then $\mu[a] =< i_1, i >$ where $1 < i_1 \leq i$ on line 16 of Algorithm 3.

Proof:

We know that $E = \bigcup_{j=1}^{i} E_j$ and $N = \bigcup_{j=1}^{i} N_j$ by definition. We also know that $a \in_\Phi E$ is equivalent to $a \in E_i$ since the $\in_\Phi$ operator ensures that the edge is only compared with edges that are incident to nodes in the Φ_i node mapping *(see Definitions above)*. Thus, let a be an edge where $a \in E_i$ and $a \in E_{i+1}$. Furthermore, given μ_i, let $\mu_i[a] \neq \emptyset$. We show that $\mu_i[a] =< i_1, i >$ where $1 \leq i_1 \leq i$ by induction.

Case (i = 0):

$E = \emptyset$ and $N = \emptyset$. When the algorithm is finished $\forall a \in E_1$, we have $E = E_1$ and $(\forall a \in E)(\mu_1[a] = \emptyset)$

Case (i = 1):

$E = E_1$ from the Case (i=0) and by necessity $N = N_1$. Let a be an edge where $a \in E_2$ and $a \in E_1$. From the Base Case we know that $\mu_1[a] = \emptyset$. Thus, it is never the case that $\mu_1[a] \neq \emptyset$ when $a \in E_2$ and $a \in E_1$ for $i = 1$.

Case (i = 2):

$E = E_2$ from the Case (i = 1) and by necessity $N = N_2$. Let a be an edge where $a \in E_3$ and $a \in E_2$. Furthermore, let $\mu_2[a] \neq \emptyset$ and $a = (n_1, n_2)$. Assume that $a \notin E_1$. Then since $\mu_2[a] \neq \emptyset$ it must be the case that $(n_1 \in N_1 \vee n_2 \in N_1)$ and $\mu_2[a] =< 2, 2 >$. If $(n_1 \notin N_1 \wedge n_2 \notin N_1)$ then $a \in E_1$ since $\mu_2[a] \neq \emptyset$. But that would imply that $\mu_1[a] \neq \emptyset$ which is never the case when $a \in E_2$ and $a \in E_1$. Therefore, it is always the case that $a \notin E_1$. Therefore, if $a \in E_3$ and $a \in E_2$ and $\mu_2[a] \neq \emptyset$ then $\mu_2[a] =< 2, 2 >$.

Base Case (i = 3):

$E = E_3$ from the Case (i = 2) and by necessity $N = N_3$. Let a be an edge where $a \in E_4$ and $a \in E_3$. Furthermore, let $\mu_3[a] \neq \emptyset$ and $a = (n_1, n_2)$. Assume that $a \notin E_2$. Then since $\mu_3[a] \neq \emptyset$ it must be the case that $(n_1 \in N_2 \vee n_2 \in N_2)$ and $\mu_3[a] =< 3, 3 >$. If $(n_1 \notin N_2 \wedge n_2 \notin N_2)$ then $a \in E_2$ since $\mu_3[a] \neq \emptyset$. Assume that $a \in E_2$. Then it must be the case that $\mu_2[a] \neq \emptyset$. Thus, from Case (i=2) we have that if $a \in E_3$ and $a \in E_2$ and $\mu_2[a] \neq \emptyset$, then $\mu_3[a] =< 2, 3 >$. Therefore, if $a \in E_4$ and $a \in E_3$ and $\mu_3[a] \neq \emptyset$ then $\mu_3[a] =< 3, 3 >$ or $\mu_3[a] =< 2, 3 >$.

Case (i+1):

$E = E_i$ from the Case (i) and by necessity $N = N_i$. Let a be an edge where $a \in E_{i+1}$ and $a \in E_i$. Furthermore, let $\mu_i[a] \neq \emptyset$ and $a = (n_1, n_2)$. Assume that $a \notin E_{i-1}$. Then since $\mu_i[a] \neq \emptyset$ it must be the case that $(n_1 \in N_{i-1} \vee n_2 \in N_{i-1})$ and $\mu_i[a] =< i, i >$. Assume that $a \in E_{i-1}$. Then it must be the case that $\mu_{i-1}[a] \neq \emptyset$. Thus, from our base case we have $\mu_i[a] =< i_1, i >$.