Flow Graph Extraction for Modular Verification of Java Programs

AFSHIN AMIGHI

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2011
Flow Graph Extraction for Modular Verification of Java Programs

AFSHIN AMIGHI

Master’s Thesis in Computer Science (30 ECTS credits)
at the Software Engineering of Distributed Systems Master’s Program
Royal Institute of Technology year 2011
Supervisor at CSC was Dilian Gurov
Examiner was Johan Håstad

TRITA-CSC-E 2011:038
ISRN-KTH/CSC/E--11/038--SE
ISSN-1653-5715
Abstract

The starting point for the project is a framework for compositional program verification based on program flow graphs, an abstraction of program control flow giving rise to an over-approximation of the source code behavior. Flow graph extraction for modular verification should allow the independent extraction of flow graphs of subsystems or modules. Furthermore the composition of the flow graphs of the modules should give a safe approximation of the complete program flow graph. The existing tools for flow graph extraction are not flexible enough for modular purposes, since they typically assume that they are given a complete program.

The goal of this study is the formal definition and implementation of modular flow graph extraction. In this project a formal translation from Java programs to target flow graph is specified. Then based on an operational semantics for the source language and for flow graphs, the correctness of the translation is proved. Flow graph extraction has to respect the modularity of programs, which is the main contribution of the work. Finally, a tool is developed based on specification of the translation.
Acknowledgments

I am indebted to my supervisor Dr. Dilian Gurov who always guided me in all my path. I am also grateful to Dr. Marieke Huisman who provided me a chance to have a visit in FMT group at University of Twente. I should thank to Prof. Arend Rensik who advised us in Groove training period and Eduardo Zambon for training me Groove during my visit at FMT group and kindly answering all my questions. I thank to Siavash Soleimanifard who evaluated the tool in his real experiments and was patient on errors, and to Pedro de Carvalho Gomes for his helpful comments about my report, and Laurent Hubert who always kindly answered my questions about Sawja.

Finally I would like to dedicate my heartfelt appreciation to my beloved wife, Elham, who was always with me in all occasions with her supports, my parents, Reza and Simin, my sisters, Noushin and Farin who always supported and encouraged me.
# Contents

I  Preliminaries  

1  Introduction  
  1.1  Problem Statement  
  1.2  Background  
    1.2.1  Key Concepts  
    1.2.2  A Program Model for Compositional Verification  
  1.3  Contribution  
  1.4  Organization  

II  Static Analysis of Program Code  

2  Java Byte-Code Analysis  
  2.1  Introduction  
  2.2  Java Byte-Code  
    2.2.1  Methods in JBC  
    2.2.2  Formal Framework for JBC  

3  Flow Graph in JBC: Formal Study  
  3.1  Introduction  
  3.2  JVM as a Transition System  
    3.2.1  Auxiliary Functions  
    3.2.2  JBC Transition Rules  
  3.3  CFG Construction  
    3.3.1  Source Code Labeling  
    3.3.2  CFG Nodes  
    3.3.3  CFG Extraction Rules  
    3.3.4  CFG as a Transition System  
  3.4  CFG Correctness  

4  Modular Flow Graph Extraction  
  4.1  Introduction  
  4.2  Modular Control Flow Graph Extraction Methods  
    4.2.1  Modular Static Program Analysis
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2 Modular Java Programs</td>
<td>59</td>
</tr>
<tr>
<td>4.2.3 Simplification-based Compositional CFG Extraction</td>
<td>59</td>
</tr>
<tr>
<td>4.2.4 Worst-case Compositional CFG Extraction</td>
<td>60</td>
</tr>
<tr>
<td>4.2.5 Compositional CFG Extraction: User-Provided Interfaces</td>
<td>67</td>
</tr>
<tr>
<td>4.2.6 Compositional CFG extraction: Symbolic Execution</td>
<td>68</td>
</tr>
<tr>
<td>4.3 Compositional CFG extraction: Hybrid Approach</td>
<td>68</td>
</tr>
<tr>
<td>5 Flow Graph in JBC: Implementation</td>
<td>71</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>71</td>
</tr>
<tr>
<td>5.2 CFG Extractor</td>
<td>72</td>
</tr>
<tr>
<td>5.3 Complete Program CFG Extractor</td>
<td>74</td>
</tr>
<tr>
<td>5.4 Modular CFG Extractor</td>
<td>78</td>
</tr>
<tr>
<td>6 Conclusions: Static Analysis</td>
<td>81</td>
</tr>
<tr>
<td>6.1 Related Work</td>
<td>81</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>83</td>
</tr>
<tr>
<td>6.3 Summary</td>
<td>84</td>
</tr>
<tr>
<td>III Graph Transformation</td>
<td>85</td>
</tr>
<tr>
<td>7 Graph Transformation</td>
<td>87</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>87</td>
</tr>
<tr>
<td>7.2 Background</td>
<td>88</td>
</tr>
<tr>
<td>7.3 GROOVE</td>
<td>90</td>
</tr>
<tr>
<td>8 CFG Extraction using Groove</td>
<td>93</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>93</td>
</tr>
<tr>
<td>8.2 A Production System for CFG Extraction</td>
<td>93</td>
</tr>
<tr>
<td>8.2.1 Basic CFG</td>
<td>94</td>
</tr>
<tr>
<td>8.2.2 Method Calls</td>
<td>98</td>
</tr>
<tr>
<td>8.3 Virtual Method Call Resolution Rules</td>
<td>99</td>
</tr>
<tr>
<td>8.3.1 Class Hierarchy Analysis - CHA</td>
<td>100</td>
</tr>
<tr>
<td>8.3.2 Rapid Type Analysis - RTA</td>
<td>101</td>
</tr>
<tr>
<td>9 Conclusions: Graph Transformation</td>
<td>105</td>
</tr>
<tr>
<td>9.1 Related Work</td>
<td>105</td>
</tr>
<tr>
<td>9.2 Future Work</td>
<td>105</td>
</tr>
<tr>
<td>Bibliography</td>
<td>107</td>
</tr>
</tbody>
</table>
Part I

Preliminaries
Chapter 1

Introduction

Requirements satisfaction plays an indispensable role in successful software development. Software verification is the process of determining whether the product fulfills its requirements stated in terms of a formal specification. In this process formal techniques are widely utilized to prove or disprove the correctness of an underlying system with respect to its formal specification.

Component based development establishes standards that enable software components to be reusable and customizable. As a consequence applications can be built using off-the-shelf units. Furthermore, in a dynamic environment, component technology enables components communicating with each other to join or leave the system dynamically, which may result in automatic self-reconfiguration of the system. Any change in components or reconfiguration of the system requires ensuring that the global system behavior is not influenced negatively.

The static and dynamic variability in non-monolithic applications and dynamic environments necessitate modular verification techniques to guarantee the correctness of the system behavior. In these techniques properties of main system are decomposed into properties of its parts such that the composition of local properties implies the global specification. Thus the process of verification turns into the verification of each part against its local properties. If the process can deduce satisfiability of each part against local properties, satisfaction of the complete system with respect to its specification is concluded.

A proper model of the system to be verified is a starting point of every verification process. Program dependence graphs (PDG), data dependency graphs, call graphs and control flow graphs are some of the well-known program models are based on the notion of graph. Control flow graphs as an intermediate representation of a program is a widely used model in different software engineering tasks. Control flow analysis proposes techniques to codify relations between control points of the program in terms of control flow graph or call graph. Static and dynamic program analysis studies different algorithms of control flow analysis.

The following study as a master degree project investigates methods and techniques for control flow graph (CFG) extraction. Compositional verification of Java
programs will be the main application area where the proposed methods can be helpful.

1.1 Problem Statement

In general, program model extraction has to conform to the verification method used. Employing modular analysis techniques to obtain the model is mandatory for compositional verification algorithms. Although there are lots of studies in control flow analysis of programs, there is a lack of formally defined and provably correct rules for CFG construction serving modular verification techniques. This is the main motivation behind this thesis.

Gurov et al. in their research propose a technique for compositional verification of safe sequences of procedure invocations [1]. The program model used in the study is control flow graphs [2]. To support this technique a tool set [3] has been developed. The starting point of this tool set is a Program Analyzer (PA), developed in Java using the Soot [4] framework, that given a set Java byte-code classes as input, extracts a program model.

Performance, scalability and conformity with modular programs are central issues of current PA. The project focuses on the analysis of Java programs and aims to achieve the following goals:

1. Formal definition of control flow graph extraction.
2. Proof of correctness of the extracted program models.
3. A control flow graph extractor tool for Java programs (to replace the current PA).
4. A study and proposal of compositional control flow graph extraction techniques for modular Java programs.

1.2 Background

1.2.1 Key Concepts

Our study relies on concepts of program compositional analysis and control flow graphs. Notions such as program analysis, modular programs, control flow graphs and virtual method calls are central to us and we give the needed background in this chapter.

Program Analysis

Program analysis is the process of automatically examining the behavior of the programs. Static code analysis studies programs properties without actually executing it. Dynamic program analysis investigates the properties of instrumented programs
1.2. BACKGROUND

by executing on a real or virtual processor. Although the analysis does not give precise information it may still give useful information. All program analyses should be semantics based: this means that information obtained from the analysis can be proved to be safe (or correct) with respect to the semantics of the programming language [7].

Control Flow Graph

Usually analysis of a property in the program is based on knowledge of the control flow of the program. Control flow analysis employs program analysis techniques to discover the control structure of the program and represent its flow of control. Control flow analysis practically originated in compilers to produce optimized programs. One of the first representations of control flow is suggested in [5], in which connectivity matrices is used. Lots of notions have been developed in this area. The main motivation of all these works is to extract and represent flow relationships in a program. Allen [6] was one of the pioneers in using graph for control flow analysis representations.

So in general control flow graph (CFG) of a program is a graph based representation of all possible paths traversed during execution. Each node of CFG is a basic block of program. A basic block in a program is a sequence of instructions with single entry point and single exit point. If a sequence of executions enters into the basic block through its entry point, all basic block instructions will be executed and execution control will pass through exit point. The granularity of this basic block for program analysis depends on the purpose of the analyzer.

Call graph is a different concept. Usually call graph refers to a model depicts a static caller-callee relation between procedures or methods in a program. Both control flow graph and call graph can be artifacts of a control flow analysis.

Modular Program Analysis

A program is modular if its source code is decomposed into several source units called modules. Every module is known via its interface for the rest of the program. Interface of the module defines exported and imported elements. Generally modules are referred to a logically self-contained and discrete unit of a larger program with a defined functionality. Sub-routines, functions, classes can be considered as modules of a program.

Compositional static Java program analysis inspects the techniques in which each class is analyzed separately. So the semantics of the whole program can be obtained compositionally from the semantics of classes. Modular control flow analysis of a Java program investigates techniques to obtain the whole programs control flow graph compositionally from control flow graphs of each class.
Virtual Methods

In any hierarchical design of an object oriented system, subclasses inherit all features of its superclass(es). Moreover they might contain some additional fields or methods to implement their specified functionality. In a class hierarchy, when a method in a subclass has the same signature\(^1\) as a method in its superclass, then the method in the subclass is said to override the method in the superclass. Thus in object oriented programming language, if a method’s behavior can be overridden within an inheriting class(es), it is called a virtual method.

Java as an object oriented programming language (OOPL) obligates every class of the program to be part of a hierarchy rooted by \texttt{java.lang.Object}. In Java, all non-static methods are by default virtual methods. Only methods marked with the keyword \texttt{final}, are not allowed to be overridden. Figure 1.1 illustrate a code fragment of using a virtual call. Variable declared in method \texttt{main} can have both types \texttt{P} or \texttt{C}.

```java
class P {
    public void m() { ... }
}
class C extends P{
    public void m() { ... }
}
class D extends P{
    public void m() { ... }
}
class Main{
    public static void main(...){
        P obj = null;
        ...
        if(...) obj = new P();
        else obj = new C();
        ...
        obj.m(); // Polymorphic call site
    }
}
```

\textbf{Figure 1.1.} Example: Virtual Method Call

If a method call can have more than one possible receiver, it is called a polymorphic call site; otherwise if the receiver can take only one type then it is called polymorphic call site; otherwise if the receiver can take only one type then it is called.

\(^1\)Signature of a method refers to the set of its return type, methods name and sequence of types of its arguments.
1.2. BACKGROUND

a monomorphic call site. Resolution is the process of determining of the receiver’s type statically. Code fragment in figure 1.1 is an example of polymorphic call site which cannot be resolved at compile time. Java compilers employs dynamic binding to resolve the type of obj.

Virtual Method Call Resolution

Compilers and program analyzers use static code analysis techniques to discover possible receivers of a virtual call without executing the program. So an object type determination in virtual call site helps to optimize the program in compile time. In control flow analysis of a given program the main objective of virtual calls resolution is to reduce the number of call edges, so resulting in a more precise control flow graph.

There are several algorithms to approximate which object(s) will be the receiver(s) of the message at run-time. We are not going to study these algorithms in detail. [8] is suggested as a reference for different practical virtual method call resolution algorithms in Java. [9] compares different algorithms of virtual method calls with respect to their cost and accuracy. In general all algorithms can be defined as a function $R_\alpha$ which given a call site as a pair of object and method signature, $(o, m)$, determines set of possible receiver types, employing algorithm $\alpha$. In following we are going to have a short explanation about two basic and well-known algorithms which we will use later in our CFG extraction methods.

Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) [10] combines objects statically defined type and inheritance relationships of a program to discover possible receivers of method call. In Java a variant of CHA takes advantages of the inheritance relationship information and estimates possible receivers to build a conservative set of edges to possible calls.

Simply, CHA lists all sub-classes of the class from which called object is created. Then it looks for classes that contain overridden method with the same signature of invocation target. If a class does not have such a method then removes it from the list. Final list will be possible candidates for call.

Let $\text{METH}$ be set of methods signatures and $\text{CLASS}$ be set of classes in a program $P$. Given a call site $o.m()$ in the body of method $M \in \text{METH}$, $\text{staticT}(o)$ gives the statically defined type$^2$ of an object $o$, say $C$, $\text{subT}(C)$ specifies the set of all derived types of $C$, say $\{T_1, T_2, ..., T_n\}$. Function $\text{sig}(m)$ specifies signature of a given method named $m$ and $\text{staticLookup}(T, m)$ looks for method named $m$ in class $T$ and if there is such method, returns its signature.

The algorithm looks for methods in all subtypes using $\text{staticLookup}(T_i, m)$ and if the signature of method $m()$ is equal with the signature of $\text{staticLookup}(T_i, m)$ (if any), then subtype $T_i$ is added to the set of reachable targets as a possible receiver.

---

$^2$An object’s static type is the object’s declaration type.
Following gives a formal definition of $\mathcal{R}_{CHA}$ for all $o.m(\ldots) \in Body(M)$:
\[
\mathcal{R}_{CHA}(o, m) = \{ T \in \text{Class} \mid \text{staticLookup}(T, m) = \text{Sig}(m) \land T \in \text{subT}(\text{staticT}(o)) \}
\]

It can be seen that CHA is a flow-insensitive algorithm and only consider statical definition of classes and their hierarchical information. If we apply CHA to our example in figure 1.1 we can obtain possible receivers with following elements:
\[
\mathcal{R}_{CHA}(obj, m) = \{ P, C, D \}
\]

**Rapid Type Analysis**

CHA takes advantage of class-inheritance relationships to specify possible callees even though there might not be any instances of some types in the hierarchy. For example in our example of figure 1.1, no object of type $D$ is instantiated, so there is no way that class $D$ can be a receiver. In an extension of CHA, class-instantiation information can be taken into account. RTA (Rapid Type Analysis) [11], analyzes type creations and prune those types from CHA reachable set which are not instantiated in reachable methods of program.

RTA makes a set of types which are instantiated in reachable methods of the program. Then starts to remove those types not in instantiated-class set from CHA constructed reachable types set. Statement `new T()` instantiates an object of type $T$ in a method $M$ of a program $P$. If we assume method `main()` as the starting point of the program $P$ and $R_{Meth}$ is the set of all methods which are reachable from `main()` then for all $M \in R_{Meth}$ we can define possible receivers in call site $o.m(\ldots) \in Body(M)$ using algorithm RTA as follows:
\[
\mathcal{R}_{RTA}(o, m) = \{ T \in \text{Class} \mid \text{new T()} \in Body(M) \land T \in \mathcal{R}_{CHA}(o, m) \}
\]

RTA also is a flow-insensitive algorithm. If we apply this algorithm to our example in figure 1.1 then we can remove type $D$ from our possible receivers. So for our example we have:
\[
\mathcal{R}_{RTA}(obj, m) = \{ P, C \}
\]

### 1.2.2 A Program Model for Compositional Verification

In the software development process there are cases where some components are not yet implemented or are only available via their specifications\(^3\). Systems where some components are given through their implementation and others through their specification are named *open systems*. A system that is developed to execute mobile code can be considered as an open system. The mobile code component is supposed to be plugged in at the time of execution. *Closed systems* address systems where all components are available in form of their implementation.

\(^3\)Specification refers to any formally expressed property of a component.
1.2. BACKGROUND

In modular verification the global correctness of the system is relativized to the correctness of its components. Actually this relativization allows the implementation of components to be changed over time (say, be optimized for performance or additional functionality) when the specifications are kept unchanged. The complete system verification process only checks the local correctness of the component in any alternation of a component’s implementation.

Gurov et al. use the general notion of model and both control flow graph structure and behavior are defined in terms of such models [2]. Here we take basic the definitions from [2] and [1]. In this thesis we define our control flow graphs based on these notions.

**Definition 1 (Model, Specification)** A model is called a (Kripke) structure \( M = (S, L, \rightarrow, A, \lambda) \) where \( S \) is a set of states, \( L \) a set of labels, \( \rightarrow \subseteq L \times S \times L \) a labeled transition relation, \( A \) a set of atomic propositions, and \( \lambda : S \rightarrow \mathcal{P}(A) \) a valuation, assigning to each state \( s \in S \) the set of atomic propositions that hold in \( s \). A specification\(^4\) \( S \) is a pair \((M, \mathbb{E})\) with \( M \) a model and \( \mathbb{E} \subseteq S \) a set of entry states.

The reachable part of a specification \( S = (M, \mathbb{E}) \) is defined by \( \mathcal{R}(S) = (M', \mathbb{E}) \), where \( M' \) is obtained from \( M \) by deleting all states and transitions not reachable from any entry state in \( \mathbb{E} \). Following shows a simple example of specification.

**Example 1 (Specification)** Figure 1.2 shows the graphical representation of the specification \( S = (M, \mathbb{E}) \), where

- \( M = (\{s_1, s_2, s_3\}, \{a, \epsilon\}, \rightarrow, \{p, q\}, \{s_1 \mapsto \{p, q\}, s_2 \mapsto \{p\}, s_3 \mapsto \emptyset\}) \)
- \( \rightarrow = \{(s_1, \epsilon, s_2), (s_2, a, s_1), (s_2, a, s_3), (s_3, a, s_1), (s_3, \epsilon, s_2)\} \)
- \( \mathbb{E} = \{s_1, s_2\} \)

As usual, entry states are depicted through additional incoming edges without source.

![Figure 1.2. Specification \( S = (M, \mathbb{E}) \) \[1\] ](image-url)

\(^4\)Actually this term should not be confused with program specifications. In fact here specification is equivalent with Initialized model.
Method specifications are the basic building blocks of flow graphs. For the basic program model for sequential programs with procedures, method specifications are defined as below.

**Definition 2 (Basic Method Specification)** A method graph for \( m \in \text{Meth} \) is a finite model \( \mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m) \), where \( V_m \) is the set of control nodes of \( m \), \( L_m = M \cup \{e\} \), \( A_m = \{m, r\} \), and \( m \in \lambda_m(v) \) for all \( v \in V_m \) (i.e., each node is tagged with the method name). A method specification for \( m \in \text{Meth} \) is a specification \((\mathcal{M}_m, E_m)\) such that \( \mathcal{M}_m \) is a method graph for \( m \) and \( E_m \subseteq V_m \) a non-empty set of entry points of \( m \).

A node \( v \in V_m \) is marked with an atomic proposition \( r \) to indicate it as a return node of the method. So if \( r \in \lambda_m(v) \) then \( v \) will be the last control point of the method. Every flow graph is equipped with an interface to define provided and required methods to its environment. The interface of a flow graph is defined as follows.

**Definition 3 (Flow Graph Interface)** A flow graph interface is a triple \( I = (I^+, I^-, C) \), where \( I^+, I^- \subseteq \text{Meth} \) are finite sets of names of provided and required methods, and \( C \subseteq \text{ContVal} \) is a finite set of control values, respectively. If \( I^- \subseteq I^+ \) then \( I \) is defined as closed.

Flow graph of a program is the disjoint union of the all flow graph of the methods defined in the program.

**Definition 4 (Flow Graph Structure)** Flow graph \( \mathcal{G} \) with interface \( I \), written \( \mathcal{G} : I \) is inductively defined by:

- \((M_m, E_m) : (\{m\}, M - \{m\}, C)\) if \((M_m, E_m)\) is a method specification for \( m \) over \( M \) and \( C \),
- \( \mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2 \) if \( \mathcal{G}_1 : I_1 \) and \( \mathcal{G}_2 : I_2 \).

**Example 2 (Method Graph)** Figure 1.3 shows a sample flow graph and its corresponding Java code.

Method specifications with exceptions are very similar to the basic method specifications, except that exceptions are added as atomic propositions. Following defines method specification with exceptions.

**Definition 5 (Method Specification with Exceptions)** A flow graph with exceptions for \( m \in \text{Meth} \) over sets \( M \subseteq \text{Meth} \) and \( E \subseteq \text{Excp} \) is a finite model \( \mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m) \) with \( V_m \) the set of control nodes of \( m \), \( L_m = M \cup \{e\} \), \( A_m = \{m, r\} \cup E \), \( m \in \lambda_m(v) \) for all \( v \in V_m \), and for all \( x, x' \in E \), if \( \{x, x'\} \subseteq \lambda_m(v) \) then \( x = x' \), i.e., each control point is tagged with at most one exception. A
public class Number {
    public static boolean even(int n) {
        if (n == 0)
            return true;
        else
            return odd(n - 1);
    }
    public static boolean odd(int n) {
        if (n == 0)
            return false;
        else
            return even(n - 1);
    }
}

method specification with exceptions for \( m \in \text{METH} \) over \( M \) and \( E \) is a specification \((\mathcal{M}_m, \mathcal{E}_m)\) s.t. \( \mathcal{M}_m \) is a flow graph with exceptions for \( m \) over \( M \) and \( \mathcal{E}_m \subseteq V_M \) a non-empty set of entry points of \( m \).

The behavior of the flow graph for the basic model (without exceptions) in closed systems is defined as follows. In this definition \( m_1 \text{ call } m_2 \) denotes an invocation from \( m_1 \) to \( m_2 \) which causes a call transition between two control nodes. \( m_1 \text{ ret } m_2 \) shows normal return from a method call which causes ret transition and finally label \( \tau \) is used for all internal normal transitions.

**Definition 6 (Basic Closed Systems Flow Graph Behavior)** Let \( G = (M, E) : (I^+, I^-) \) be a closed flow graph such that \( M = (V, L, \rightarrow, A, \lambda) \). The behavior of \( G \) is described by the specification \( b(G) = (M_b, E_b) \), where \( M_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b) \), s.t. \( S_b = V \times V^* \), that is, states (or configurations) are pairs of control points \( V \) and stacks \( \sigma \), \( L_b = \{ m_1 \text{ l } m_2 | l \in \{ \text{call, ret} \}, m_1, m_2 \in I^+ \} \cup \{ \tau \} \), \( A_b = A \), \( \lambda_b((v, \sigma)) = \lambda(v) \), and \( \rightarrow_b \subseteq S_b \times L_b \times S_b \) is defined as follows:

\[
\begin{align*}
[\text{transfer}] \quad & (v, \sigma) \xrightarrow{r_b} (v', \sigma) \\
& m \in I^+, \ v \xrightarrow{m} v', \ v \Vdash \neg r
\end{align*}
\]

\[
\begin{align*}
[\text{call}] \quad & (v_1, \sigma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v'_1, \sigma) \\
& \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2} v'_1, v_1 \Vdash \neg r, v_2 \Vdash m_2, v_2 \in E
\end{align*}
\]

\[
\begin{align*}
[\text{return}] \quad & (v_2, v_1, \sigma) \xrightarrow{m_2 \text{ ret } m_1} (v_1, \sigma) \\
& \text{if } m_1, m_2 \in I^+, v_2 \Vdash m_2 \land r, v_1 \Vdash m_1
\end{align*}
\]

The set of initial configurations is defined by \( E_b = E \times \{ \epsilon \} \), where \( \epsilon \) denotes the empty sequence over \( V \).

In an open system, we need to define the transitions for calls to unavailable components. Following extends definition 6 with caret rule. Intuitively this rule
assumes the call terminates and the execution control passes to the next instruction of the caller. The following defines the behavior of an open system flow graph formally as a generalization of definition 6.

**Definition 7 (Basic Open Systems Flow Graph Behavior [13])** Let \( G = (M, E) \) be an open flow graph such that \( M = (V, L, \to, A, \lambda) \). The behavior of \( G \) is described by the specification \( b(G) = (M_b, E_b) \), where \( M_b = (S_b, L_b, \to_b, A_b, \lambda_b) \), s.t. \( S_b = V \times V^* \), that is, states (or configurations) are pairs of control points \( V \) and stacks \( \sigma \), \( L_b = \{ m_1 l m_2 \mid l \in \{ \text{call, ret} \}, m_1, m_2 \in I^+ \} \cup \{ m_1 \text{ caret} m_2 \mid m_1 \in I^+ \land m_2 \in I^- \} \cup \{ \tau \} \), \( A_b = A \), \( \lambda_b((v, \sigma)) = \lambda(v) \), and \( \to_b \subseteq S_b \times L_b \times S_b \) is defined as follows:

1. **[transfer]** \((v, \sigma) \xrightarrow{\tau} (v', \sigma)\)
   \(m \in I^+, v \xrightarrow{\ell} m v', v \models \neg r\)

2. **[call]** \((v_1, \sigma) \xrightarrow{m_1 \text{ call} m_2} (v_2, v_1, \sigma)\)
   \(m_1, m_2 \in I^+, v_1 \xrightarrow{m_2} m_1 v_1', v_1 \models \neg r, v_2 \models m_2, v_2 \in E\)

3. **[return]** \((v_2, v_1, \sigma) \xrightarrow{m_2 \text{ ret} m_1} (v_1, \sigma)\)
   \(m_1, m_2 \in I^+, v_2 \models m_2 \land r, v_1 \models m_1\)

4. **[caret]** \((v_1, \sigma) \xrightarrow{m_1 \text{ caret} m_2} (v_1', \sigma)\)
   \(m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2} m_1 v_1', v_1 \models \neg r, v_1' \models m_1\)

The set of initial configurations is defined by \( E_b = E \times \{ \epsilon \} \), where \( \epsilon \) denotes the empty sequence over \( V \).

Example 3 taken from [13] shows the behavior of flow graph represented in example 2.

**Example 3 (Flow Graph Behavior)** In even-odd flow graph of figure 1.3, one example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

\[
(v_0, \epsilon) \xrightarrow{\tau} (v_1, \epsilon) \xrightarrow{\text{even call odd}} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\text{odd ret even}} (v_8, v_3) \xrightarrow{\tau} (v_3, \epsilon)
\]

Now, consider just the method graph of method even as an open flow graph, having interface \( \{ \text{even} \}, \{ \text{odd} \} \). The local contribution of method even to the above global behavior is the following run:

\[
(v_0, \epsilon) \xrightarrow{\tau} (v_1, \epsilon) \xrightarrow{\text{even caret odd}} (v_3, \epsilon)
\]
1.3 Contribution

I use two different methods to accomplish the control flow graph extraction of Java programs, the first based on static analysis the second on graph transformation.

The main tasks in the first approach are:

- Formal definition of control flow graph extraction based on the operational semantics of the Java byte-code instructions.
- Formal proof of control flow graph correctness in terms of a simulation relation.
- Formal study of modular control flow graph extraction using compositional analysis for modular Java programs.
- Implementation of CFG extraction in the OCaml programming language.

In the second approach I employ a graph transformation technique to extract basic (non-exceptional) control flow graphs. The tasks can be listed as follows:

- Graph re-writing rules for CHA and RTA algorithms to resolve virtual method calls.
- Graph re-writing rules for basic (non-exceptional) control flow graph extraction.

Further, I summarized the operational semantics of Java byte-code given in [18] and re-wrote them using functions in 3.2.1 defined by myself. I formalized the format of nodes in Definition 9. The CFG formal construction rules defined in 3.3.3 are due to me. The basic transformation in Definition 13 is formed in common discussion with my supervisor. I extended the definition with exceptions.

In Definition 12 I extended the definition of open system flow graph exceptional behavior with exceptional return rule $\texttt{xreturn}$. I am responsible for the proof of Theorem 1 where I proved that the extracted CFG simulates the original Java byte-code program. I adapted the modular control flow graph extraction rules with compositional separate code analysis techniques by myself.

In the implementation part, the development of two versions of CFG extractor for Java byte-code programs is my work. I reused some modules of the Sawja framework [30] for parsing incomplete programs with some minor changes.

In the graph transformation technique, the rules for method call bindings are defined by myself. Type graphs for our program model are defined in a joint work with Eduardo Zambon during a two-day visit of the FMT group in University of Twente. The rules to extract CFG based on our program model are due to me.

1.4 Organization

The study is reported in three parts. The first part contains introduction and basic background required in proceeding chapters. In order to have knowledge about the
problem statement and basic definitions part 1 needs to be studied. Having read the prerequisites, second and third part explains two different approaches selected to solve the problem. There is no dependency between part 2 and 3.

The second part contains five chapters about programs code static analysis, the technique used to extract control flow graphs from the programs given in Java byte-code. Required knowledge about this static analysis approach is explained in chapter 2. Chapter 3 studies control flow graph extraction from Java byte-code programs formally. Formal definition of the problem, CFG extraction rules using JBC semantics, correctness of the result CFG. Implementation in OCaml language of CFG extractor will be explained in chapter 4. Modular flow graph extraction is topic of chapter 5. Related work in CFG extraction using code static analysis along with future work will be given in chapter 6.

Part 3 contains chapters of graph transformation technique used to extract CFG of Java source code. Chapter 7 includes required background for this approach. Rules defined to extract CFG of a Java source code with the emphasis on virtual method call resolution algorithms are described in chapter 8. Related work in CFG extraction using graph transformation techniques along with future work will be given in chapter 9. Figure 1.4 shows dependencies between the chapters of the report.

![Figure 1.4. Report Chapters Dependencies](image-url)
Part II

Static Analysis of Program Code
Chapter 2

Java Byte-Code Analysis

Static analysis of program code is one of the methods used to extract control flow graphs from Java programs. Here we focus at the byte-code level of the programs. This chapter provides a background about static program analysis, Java byte-code and the basic required theoretical definitions.

2.1 Introduction

Program code static analysis is defined as inferring facts about the behavior of the program without actually executing it. Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer [7]. Compilers, program testing and verification are major application areas for static analysis techniques.

Java source code typically is translated into an intermediate language of instructions named Java Byte-Code. The byte-code is executed by Java virtual machine. Byte-code is an important option for mobile code applications\(^1\). Because mobile code is not always trusted, byte-code verification becomes crucial. Techniques of program code static analysis assist the verification of properties in Java applications. A subset of security properties is verified by byte-code verifier to prevent a range of possible run-time errors.

Furthermore verification of the program against its specification usually requires a correct model of the program. Control flow graph as a result of control flow analysis is a common program model employed by verification techniques.

In compositional verification techniques developed by Gurov et al. safety properties of inter-procedural control flow, i.e. properties describing safe sequences of procedure invocations are analyzed. Programs in Java byte-code format are considered as the input of the verification process. The techniques are applicable in any context concerned with inter-procedural control-flow properties of components.

\(^1\)Mobile code is referred to software transferred between systems and executed on a local system without explicit installations. Java Applets can be exemplified as mobile code.
communicating via procedure calls [1]. There are different compilers developed to translate other programming languages like Ada, Scala and C into Java byte-code. Therefore static analysis of Java programs in byte-code level gives the opportunity to apply the techniques in a wide range of applications implemented in different languages. Furthermore a study on modular flow graph extraction will give the ability to have a unified compositional verification tool set.

Studying and analyzing control flow of Java programs require some knowledge about Java byte-code (JBC), their execution model and formal definitions. In this chapter we present a general overview of the syntax and semantics of JBC instructions. In addition, a summary of the Java virtual machine (JVM) behavior when executing JBC programs is presented. Then a formal model required to our study is explained. Finally, theoretical definition and concepts are reviewed to provide background for the next chapter, which presents the flow graph extraction process formally.

2.2 Java Byte-Code

The JVM is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time [17]. Java byte-code is the machine language of the Java Virtual Machine. Compiling each class in Java source code produces a .class file including the class declaration and its implementation code. After translation of a Java program, we will have a collection of class files organized into packages. Each class file contains JVM instructions (byte-codes) and a symbol table, as well as other auxiliary information.

2.2.1 Methods in JBC

A method code in a JBC class is a set of byte-code instructions. Each instruction consists of an operation code and operands, if any. JBC instructions provide simple stack based operations, which are executed by JVM. For example instruction push c pushes the constant c onto the stack; push is considered as opcode and c as operand. Also, pop is an instruction without any operand which simply pops the top elements off the operands stack.

Instructions

The instruction set of a method is stored in a byte-code array. The byte-code array is indexed by a program counter (pc). First instruction of a method is located in pc = 0. Each instruction of a method is labeled with the index of this array, where each opcode and its arguments are stored. The labels might not be sequential because each opcode occupies one byte and some of the opcodes have parameters that take up space in the byte-code array. For example, the aload_x opcode has no parameters and naturally occupies one byte in the byte-code array. Therefore, if one assumes that aload_x is in address i then the next opcode, will take location
2.2. JAVA BYTE-CODE

$i+1$. As another example `getfield` opcode and its parameters occupy three bytes. So the next instruction after `getfield` can be found at location $j+3$ if $j$ is assumed as the index of `getfield` opcode. Code fragment in Figure 2.1 shows a sample of Java byte-code with corresponding addresses (labels).

0: `aload_0` //Push the object reference(this)
1: `getfield#6` //Pop the object reference(this)
4: `iconst_0` //Push 0.
5: `iaload` //Pop the top two values
6: `ireturn` //Pop top value and push it on the operand stack of the invoking method. Exit.

Figure 2.1. Java byte-code instruction

Execution

Java Virtual Machine (JVM) executes a Java Byte-Code (JBC) program. A JVM is a stack-based interpreter of JBC methods. In JVM each thread has its own stack, which stores *frames*. For each method invocation, a new frame is created and completing of invocation destroys the created frame. Each method frame consists of an operand stack, an array of local variables and a reference to the runtime constant pool of the class of the current method. Parameters for each method and the values of the local variables are located in the array of local variables, also called the local variable table.

The operand stack is a LIFO stack used to push and pop values. It is used by certain opcodes to pop, manipulate values and push the results. The operand stack is also used to receive return values from callee methods. In every execution cycle, JVM uses program counter register to fetch next instruction from the byte-code array (instructions set).

JVM also uses a specific area of the memory for dynamic memory allocation (objects and arrays), called *heap*. This heap area is shared among all JVM threads. JVM creates heap on start up. Created objects are not explicitly deallocated in JBC and the garbage collector\(^2\) is responsible for cleaning the area. Figure 2.2 shows a simple model of JVM execution environment.

Exceptions

Any semantically constraint violation by instructions is reported by JVM as an exception. An attempt to index outside of an array bound or addressing a field of unallocated class instance are examples of the exceptions. A programmer can specify where the control point should be if one occurs. Exceptions also can be thrown explicitly by instruction `throw`. Each method can have handlers to catch the exceptions. If none could be found in the currently executing method, its execution

\(^2\)An automatic program to release the memory occupied by objects that are no longer in use by the program.
is completed abruptly and JVM looks for a proper handler in caller context. This process continues until a correct handler is found or thread of execution terminates exceptionally. If JVM was able to find an appropriate handler for the exception, then the execution control jumps to the first instruction specified by the handler.

**Subroutines**

A *finally* block ensures that if an unexpected exception has occurred the statements protected by *finally* block will be executed. In Java byte-code specification a Java compiler is supposed to implement the *finally* blocks using subroutines. The idea of subroutines may come because it is desirable to share the common code of a *finally* block. A subroutine can be considered as a procedure in byte-code level. Unlike method calls, subroutines share frame with their callers. Therefore, they can manipulate the same register set and stack as their callers do.

A subroutine is called by a special jump-to subroutine instruction \texttt{jsr}, which pushes the successor of the current address onto the stack. Then the subroutine has to store the current address in a (not pre-defined) register, from which it is later retrieved by a return-from-subroutine instruction \texttt{ret}.

### 2.2.2 Formal Framework for JBC

So far, we had an informal overview of Java byte-code instructions. In order to analyze the programs formally we need to have a formal specification of the language. The formal explanation of JBC presented here is taken from [18], which is a formal framework for the Java byte-code language proposed by Freund and Mitchell.

The formal framework of JBC models a JBC program \(^3\) by environment \(\Gamma\), which contains all information about classes, interfaces and methods. In fact \(\Gamma\) is a partial map from class names, interface names and method references to their respective

\(^3\)The subset of JBC which is modeled in [18] is called \textit{JVML}. 

20
2.2. JAVA BYTE-CODE

\[
\Gamma^C : ClassName \rightarrow \left\langle \begin{array}{l}
\text{super: ClassName} \cup \{\text{None}\}, \\
\text{interfaces: set of InterfaceName,} \\
\text{fields: set of FieldRef}
\end{array} \right\rangle
\]

\[
\Gamma^M : MethodRef \rightarrow \left\langle \begin{array}{l}
\text{code: Instruction}^+, \\
\text{handlers: Handler}^*
\end{array} \right\rangle
\]

\[
\Gamma^I : InterfaceName \rightarrow \left\langle \begin{array}{l}
\text{interfaces: set of InterfaceName,} \\
\text{methods: set of Interface MethodRef}
\end{array} \right\rangle
\]

\[
\Gamma = \Gamma^C \cup \Gamma^I \cup \Gamma^M
\]

Figure 2.3. Program Environment

definitions (figure 2.3). Sub-typing in an environment (program) is indicated by \( \Gamma \vdash \tau_1 < : \tau_2 \) which shows \( \tau_1 \) as a sub-type of \( \tau_2 \) in environment \( \Gamma \).

References are introduced in [18] to uniquely identify methods, interfaces and fields. So a signature (reference) carries information about the method or field in three parts: the class or interface in which it was declared, the name and the type. A method is made of a signature, local variables and body code. We need here to make a convention on method signature and name representation. Capital letters, like \( M \), is for method signature with name \( m \). As we know there might be several methods with different signature and same name. So methods signatures can be indexed to show different signatures with equal names.

Let us define \( \text{Meth} \) as a set of method names. \( m \in \text{Meth} \) in an environment \( \Gamma \) is represented as \( \Gamma[M] \) in which \( M \) denotes the signature of \( m \). Each method consists of a body code and exception handlers table which is defined as \( \Gamma[M] = \langle P, H \rangle \). If we define \( \text{ADDR} \) as all possible valid instructions' addresses in \( \Gamma \) then \( \text{Dom}(P) \subseteq \text{ADDR} \) is the set of valid program addresses for method \( M \) and \( P[k] \) shows the instruction at position \( k \in \text{Dom}(P) \) in method body code. \( 0 \in \text{Dom}(P) \) is the minimum valid address and is the location of method’s first instruction. For the sake of simplicity we will use \( M[k] = i \) to locate instruction \( i \in \text{ADDR} \) at location \( k \) of method \( M \). Exception handlers table \( H \) partially maps an interval of addresses to a specific address, which is the position of handler.

In this framework a method body code is considered as a sequential array such that each cell is occupied with opcode and its belonging parameters. Program counter is employed to fetch each instruction to be executed. So simply in order to get next instruction, program counter or index of the current instruction in instruction’s array is incremented by one. So if we assume \( k \) as the index of \text{getfield} in instruction array the next instruction will have index \( k + 1 \).

In order to have a clear notion of next instruction in our model we need to differentiate between opcode array and instruction array. Opcode array refers to the byte-code array defined by JVM standard specification and instruction array
refers to sequence of the method code indexed sequentially as defined in [18]. So if we define ADDR as a set of valid addresses in a program, next instruction can be abstracted here using \(\text{succ} : \text{ADDR} \rightarrow \text{ADDR} \) function, which is defined as follows:

\[
\text{succ } p = \begin{cases} 
  p + 1 & \text{if } p \text{ is instruction index} \\ 
  p + \text{size}(M[p]) & \text{if } p \text{ is the program pointer}
\end{cases}
\]

in which \(\text{size}(M[p])\) returns the size of the instruction in instruction array of method \(M\) at position \(p\). So regardless of whether ADDR is the set of instruction indices or program pointer addresses, \(\text{succ } p\) refers to a unique opcode or instruction.

Freund and Mitchell introduce JVM execution state based on its behavior which covers activation records containing information of the current active method, program counter, stack of operands and set of local variables and a heap as object store. In this model execution state is a configuration \(C = A; h\), where \(A\) denotes the sequence of activation records and \(h\) is for heap. Each activation record in the sequence is created in every method invocation and formally we can define the sequence as follows:

\[
A ::= A' | \langle x \rangle_{\text{exc}} A' \\
A' ::= \langle M, pc, f, s, z \rangle. A' | \epsilon
\]

in which \(M\) shows the method reference of active method, \(pc\) for program counter, \(f\) is a map from local variables to values, \(s\) denotes operand stack and finally \(z\) is initialization information for the object being initialized in a constructor. \(\langle x \rangle_{\text{exc}}\) shows the records for exception handling. In case of any exception, JVM pushes an activation record of the form \(\langle x \rangle_{\text{exc}}\) where \(x \in \text{Excp}\) denotes the exception which must be of type \(\text{Throwable}\).

In order to handle an exception, JVM seeks exception table declared in a method to find a set of instructions which are supposed to be executed in case of exceptions. As mentioned before \(H\) is a partial map for exception handling which has the form \(\langle b, e, t, \varrho \rangle\) in which \(b, e, t \in \text{ADDR}\) and \(\varrho \in \text{Excp}\). If any exception of type \(\varrho\) in program environment \(\Gamma (\Gamma \vdash x <: \varrho)\) is thrown by instructions with index \(i \in [b, e)\) then \(M[t]\) will be the first instruction of proper handler. In fact the set of instructions between \(b\) and \(e\) models the \textit{try} block. Instructions started at \(t\) models the \textit{catch} block of Java, that handles the exception.

In order to manage \textit{finally} block, a special type of exception called \textit{Any} is defined. JVM always should guarantee that instruction set from finally block should be executed. Programmer can define \textit{try-finally} block without having defined \textit{catch} block. So in order to guarantee the execution of \textit{finally} code, instructions inside the block are considered as a special handler with exception \textit{Any}. It means that if any exception happens, execution control is transferred to the address defined by handler. Better said:

\[
\forall x \in \text{Excp}, \quad \Gamma \vdash x <: \text{Any}
\]

Figure 2.4 shows a set of JBC instructions containing exception handlers for some types of exceptions.
2.2. JAVA BYTE-CODE

```
0: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc #3;       //String In try block
5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: new #5;       //class java/lang/Exception
11: dup
12: invokespecial #6; //Method java/lang/Exception."<init>";()V
15: athrow
16: astore_1
17: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
20: ldc #8;       //String In catch IOException block
22: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
25: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
28: aload_1
29: invokevirtual #9; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
32: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
35: ldc #10;      //String In finally block
37: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
40: goto 81
43: astore_1
44: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
47: ldc #11;      //String In catch Exception block
50: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
53: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
56: aload_1
59: invokevirtual #9; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
62: ldc #10;      //String In finally block
64: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
67: goto 81
70: astore_2
71: getstatic #2;  //Field java/lang/System.out:Ljava/io/PrintStream;
74: ldc #10;      //String In finally block
77: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
79: aload_2
80: athrow
81: return

Exception table:
from to target type
0  16  16 Class java/io/IOException
0  16  43 Class java/lang/Exception
0  32  70 any
43 59  70 any
70  71  70 any
```

Figure 2.4. Java byte-code instruction
Chapter 3

Flow Graph in JBC: Formal Study

The following chapter defines formally the rules for control flow graph extraction. Based on the JBC operational semantic rules and behavioral definition of corresponding flow graph we prove that the extracted flow graph simulates the Java bytecode.

3.1 Introduction

To have a sound program model, we have to prove that the model is able to follow all executions of the code. In section 2.2.2 we have introduced a formal framework of JBC. But still we have not defined how the control transitions are flowing in a set of the JBC instructions.

Operational semantics is a way that gives a statement of a program a mathematical interpretation and assists to reason formally about the way that the program executes. So operational semantics of the JBC instructions can be helpful to understand precisely how the control is flowing in the program.

On the other side we need to define how elements of a CFG are related to the execution of the code. CFG extraction rules are defined syntactically based on the program instructions set. So in structural level, a program given in Java byte-code relates to the control flow graph extracted by our formal rules. The behavior of the code is defined by the JVM executing the byte-code. Therefore we need to define the behavior of the extracted CFG as a variant of labeled transition system, thus relates the behavior of the CFG to the JVM. Figure 3.1 depicts the relationships in structural and behavioral levels.

In this chapter we set the study goal to have a provably sound CFG of a given JBC program. To achieve this we use the operational semantics of JBC instructions defined in [18] and emphasize our study on the flow of control in instructions. We define the structure and behavior of corresponding CFG as formal construction rules. Finally we prove that constructed CFG is sound.
3.2 JVM as a Transition System

In order to understand how a JBC program is executed and how JVM behaves we need to model the execution of a byte-code program as a transition system. First we need to define the rules. Each rule has some conditions that need to be satisfied. So the transition rules need to employ predefined auxiliary functions to check the application conditions. Thus here we are going to define the functions that will be used in transition rules and then the rules for each category of instructions set will be introduced.

3.2.1 Auxiliary Functions

Based on operational semantics of conditional branch instructions in [18], the transition needs to make a decision based on the operation and stack content. Function cond, given an activation record \((M, pc, f, s, z)\), checks the condition based on the instruction in \(M[pc]\). It is defined as follows for instruction \((i = \text{ifeq } q)\) and similarly can be defined for the rest of the conditional instructions.

\[
\text{cond}(⟨M, pc, f, v_1, v_2, s, z⟩) = \begin{cases} 
\text{tt} & \text{if } (M[pc] = \text{ifeq } q \land v_1 = v_2) \\
\text{ff} & \text{if } (M[pc] = \text{ifeq } q \land v_1 \neq v_2)
\end{cases}
\]

If an exception \(x\) happens at location \(p\) of method \(M\), function handle\((\Gamma, M, p, x)\) tries to find a correct handler for \(x\) in the method’s exception handlers table in a given environment \(\Gamma\). In other words if there is an entry in exception handlers table of the method for exception \(x\) thrown in program point \(p\), then handle\((\Gamma, M, p, x)\) gives the handler address, otherwise it will give zero. Transitions use this function to see if a proper handler exists for the thrown exception or not. In transition definitions, for the sake of simplicity we use \(ℏ_{\Gamma[M]}(p, x)\) instead of handle\((\Gamma, M, p, x)\). The function is defined formally as follows:

\[
h_{\Gamma[M]}(p, x) ≡ \text{handle}(\Gamma, M, p, x) = \begin{cases} 
t \neq 0 & \text{if } \exists (b, e, t, q) ∈ H_{\Gamma[M]}; p ∈ [b, e) \land \Gamma ⊢ x <: q \\
0 & \text{otherwise}
\end{cases}
\]
JVM needs to check current execution state for instruction’s constraints satisfaction. Assuming \( \text{Jvc} \) as the set of JVM configurations we define function \( \nu : \text{Jvc} \rightarrow \mathcal{P} (\text{Excp}) \) to model the check, which, given a configuration, defines all possible exceptions that might be thrown. We present the definition of this function for a subset of instructions. Others can be defined similarly.

\[
\nu((M, p, f, v, s, z)) = \begin{cases} 
\text{NullPointerExceptionExc} & \text{if } M[p]=(\text{getfield } f) \text{ and } v=\text{null} \\
\text{OutOfIndexExc} & \text{if } M[p]=(\text{newarray } a) \text{ and } v < 0
\end{cases}
\]

### 3.2.2 JBC Transition Rules

Now, based on the formal models of JBC and JVM execution configurations we are ready to define the operational semantics rules in terms of a labeled transition system. Each transition relates two execution configurations with the emphasis of execution control points. The general format for representing transitions is

\[
[R] \quad \frac{M[p] = i \in I \cdot c_0 \cdots c_n}{S \xrightarrow{\text{ } R \text{ } } S'}
\]

which is read as if instruction at point \( p \) from instruction set of method \( M \) is in \( I \subset \text{Inst} \) and conditions \( \{c_0, \cdots, c_n\} \) hold then there is a transition from execution state \( S \) to the execution state \( S' \) and the rule name is \( R \).

These transitions are defined by structural analysis on the JBC instruction set following the operational semantics defined in [18]. In fact in these rules we grouped JBC instructions by considering their flow of control. So the rules in each group do not focus on contents of operand stack, local variables and object creation. Program control transfer, activation records stack and exceptions are the one which are studied in transitions. To classify the rules, JBC instruction set is partitioned in subsets of \( \text{CmpInst} \), \( \text{CndInst} \), \( \text{JmpInst} \), \( \text{XmpInst} \), \( \text{ThrInst} \), \( \text{InvInst} \) and \( \text{RetInst} \). Following defines execution of a JBC program as a variation of labeled transition system where we classify transition rules.

**Definition 8 (JVM as Labeled Transition System with Valuations)** For a given program with environment \( \Gamma \), labeled transition system for program execution is \( \mathcal{L}_\Gamma = (V, \Sigma, \rightarrow, A, \lambda) \) where:

- \( V \): set of JVM execution states in the form of \( A;h \) in which \( A \) is the activation record \( (\langle A, 0, f, s, z \rangle) \) of the method being executed and \( h \) is the heap.
- \( \Sigma = \text{Inst} \cup \{\epsilon\} \) is the set of labels.
- \( A = \{M, e, r\} \cup \text{Excp} \) is the set of atomic propositions.
- \( \lambda : V \rightarrow \mathcal{P}(A) \) is the valuation function with following definition:

\[
\lambda(v) = \begin{cases} 
\{M, e\} & \text{if } v = \langle M, 0, f, s, z \rangle; A; h \\
\{M, r\} & \text{if } v = \langle M, pc, f, s, z \rangle; A; h \text{ and } M[pc] \in \text{RetInst} \\
\{M, x\} & \text{if } v = \langle x \rangle_{\text{exc}}, \langle M, pc, f, s, z \rangle; A; h \\
\{M\} & \text{if } v = \langle M, pc, f, s, z \rangle; A; h \text{ and } M[pc] \notin \text{RetInst} \text{ and } pc \neq 0
\end{cases}
\]
• $\rightarrow$ is the control flow transitions with following rules.

**Normal Computations**

This subset contains instructions that does not cause any exception, and after execution JVM picks next instruction at $\text{succ} \ p$. We name this set as $\text{CmpInst}$ and, the related transition rule as $[\text{Cmp}]$. Following we present some sample members of the set and the transition rule.

$$\text{CmpInst} = \{\text{add, load } r, \text{ store } r, \text{ push } v, \text{ pop}\}$$

$$[\text{Cmp}] \quad M[p] = i \in \text{CmpInst} \quad \frac{}{(M, p, f, s, z).A; h) \xrightarrow{i} ((M, \text{succ} \ p, f', s', z').A; h')$$

**Normal Conditional**

This subset contains instructions that does not cause any exception. Moreover, executing the instruction, JVM picks next instruction based on the condition either at $\text{succ} \ p$ or at $q$ which is specified by instruction, e.g. $\text{ifeq} \ q$. We name this set as $\text{CndInst}$ and we show some members of the set along with the rule.

$$\text{CndInst} = \{\text{ifeq } q, \text{ ifneq } q\}$$

$$[\text{Cnd}^{tt}] \quad M[p] = i \in \text{CndInst} \quad \text{cond}((M, p, f, s, z)) = \text{tt} \quad \frac{}{(M, p, f, s, z).A; h) \xrightarrow{i} ((M, q, f, s, z).A; h')$$

$$[\text{Cnd}^{ff}] \quad M[p] = i \in \text{CndInst} \quad \text{cond}((M, p, f, s, z)) = \text{ff} \quad \frac{}{(M, p, f, s, z).A; h) \xrightarrow{i} ((M, \text{succ} \ p, f, s, z).A; h')$$

**Unconditional Jumps**

That shows a subset of instructions that does not throw exceptions and after execution, JVM unconditionally picks next instruction at $q$, which is specified by instruction, e.g. $\text{goto} \ q$. We name this set as $\text{JmpInst}$. Note that $(jsr \ q) \notin \text{JmpInst}$. The set members and the rule is defined in below.

$$\text{JmpInst} = \{\text{goto } q\}$$

$$[\text{Jmp}] \quad M[p] = i \in \text{JmpInst} \quad \frac{}{(M, p, f, s, z).A; h) \xrightarrow{i} ((M, q, f, s, z).A; h')$$

28
3.2. JVM AS A TRANSITION SYSTEM

Exceptional Computations

This is the subset of instructions that if any semantic constraints of them are violated, an exception is thrown by JVM. If the execution is terminated normally (without any violation), JVM picks next instruction at succ pc otherwise an exceptional activation record with the type of the exception is pushed to the activation record stack and control is passed to the JVM to manage the exception. As some samples getfield, div and arraylength are kind of opcodes with possibility of exception throwing. This set is denoted as XmpInst. Sample members of the set together with the rules are as follows.

\[
\text{XmpInst} = \{ \text{div}, \text{arraylength}, \text{getfield f}, \text{putfield f}, \text{arrayload a}, \text{arraystore a}, \text{newarray a} \}
\]

\[\begin{align*}
\text{[XMP1]} & \quad M[p] = i \in \text{XmpInst} \quad \nu((M,p,f,s,z)) = \{ x : x \in \text{Excp} \} \\
& \quad (\langle M, p, f, s, z \rangle.A; h) \xrightarrow{x} (\langle x \rangle_{\text{exc}}, \langle M, p, f, s, z \rangle.A; h')
\end{align*}\]

\[\begin{align*}
\text{[XMP2]} & \quad M[p] = i \in \text{XmpInst} \quad \nu((M,p,f,s,z)) = \emptyset \\
& \quad (\langle M, p, f, s, z \rangle.A; h) \xrightarrow{x} (\langle M, \text{succ} p, f', s', z' \rangle.A; h')
\end{align*}\]

Throw Instruction

This is the subset of instructions in which elements have no normal execution. This subset has only one element throw \( \varrho \). Executing this instruction, JVM pushes an exceptional activation record with the type of the exception \( \varrho \) to the activation record stack and control is passed to the JVM to manage the exception. This single element set is denoted as ThrInst and the rule is defined as follows.

\[\text{ThrInst} = \{ \text{throw} \ \varrho \}\]

\[\begin{align*}
\text{[THR]} & \quad M[p] = i \in \text{ThrInst} \\
& \quad (\langle M, p, f, s, z \rangle.A; h) \xrightarrow{x} (\langle \varrho \rangle_{\text{exc}}, \langle M, p, f', s', z' \rangle.A; h')
\end{align*}\]

Invocations

This subset contains instructions whose passes execution control to the method which is being called. Two cases may happen when the execution of the method finishes. If the callee finishes normally then JVM picks next instruction at succ pc from caller’s instruction set. If an exceptional return happens then JVM will look for proper handler which might be in the callee’s method or callee has to terminate abruptly. We name this set as InvInst. It might be important to note that invocations are excluded from XmpInst. Although XmpInst \( \cap \) InvInst = \( \emptyset \), execution of any instruction in this subset can cause also exceptions. For example if the object variable of invokevirtual is null then an exception is thrown. Set members and the rules are defined as follows:

\[\text{InvInst} = \{ \text{invokevirtual (o,N)}, \text{invoespecial (o,N)}, \text{invokeinterface (o,N)} \}\]
CHAPTER 3. FLOW GRAPH IN JBC: FORMAL STUDY

In rule [XNV] since we excluded call instructions from XmpInst we have to define constraint violations for these instructions separately. 

Returns

This expresses subset of instructions which executions cause JVM pop current active record and control passes to the caller. Normally elements of this subset are the last instruction of the method body code. We name them RetInst. Note that ret ∉ RetInst (ret is the return instruction in subroutines which will be explained later). The corresponding rules are defined here.

Silent Instructions

When an exception happens, JVM takes the control of the execution to handle the exception. There is no instruction on JBC instructions set to accomplish handling. We call these transitions as silent transitions. We label transitions in rules [Exc_k] with ε because these transitions are performed by JVM and there is no instruction of JBC to change the configurations. The rules for the JVM silent transitions are defined as follows:
3.2. JVM AS A TRANSITION SYSTEM

Subroutines
If JBC complete instruction set is considered here, it can be found that we did not study instructions which are used in subroutines. Subroutines are reported as major complication for Java byte-code verification. They are notorious for being difficult to fit in data flow analysis based on the JVM specifications [19]. As it was explained previously, subroutines save the return point in a register which is not predefined. This creates the difficulty to discover the return points of subroutines by program static analysis. Inlining methods are proposed to simplify code analysis of subroutines [20]. There are some studies which suggest to copy finally block codes instead of using subroutines [19]. Sun’s J2SE compilers, (version 1.4.2 and later) compile finally blocks without subroutines. Therefore here we will not study this subset and we assume that there is no instruction from this set in our programs. Below shows the members of this set.

\[
\begin{align*}
SBR &= \{\text{jsr } q, \text{ ret } r\}
\end{align*}
\]

Figure 3.2 collects all the JVM transition rules. Now based on these simplified operational semantics of JBC and its behavior we are ready to list the rules for
3.3 CFG Construction

Control flow graph construction rules basically use byte-code instructions to build the elements (nodes and edges) of the graph. Rules take advantages of the instruction’s label (byte-code address) as a reference to the execution point. Then depending on the instruction at a given address current control node and next possible control nodes are constructed. Directed edges are established between the control nodes by the rules indicating possible flow of execution. The construction rules are applied syntactically using Java byte-code as input. Normally for a programming language like Java source code rules are context-sensitive, because statements contain blocks and each control point needs to know where the end of the block is. In JBC, since each instruction independently can define the source and target nodes of directed flow edge, the rules can be context-free.

Control flow graph of a given program must simulate all possible execution paths of the program. So the CFG of each method must cover all possible control points of the method. This makes the node of a CFG as a fundamental component to build. Here based on JBC programs structure and properties of the control points in our program model we define the node formally. To achieve the formal definition of the node labeling mechanism in JBC programs are explained.

3.3.1 Source Code Labeling

In general, labeling of a program statements is a fundamental step of flow graph construction. A label is defined as an explicit and unique identification assigned to a fixed position within the source code. Source code labeling is used as marking specific positions and other than marking a position a label has no effect. Depending on the programming language, different identifications can be used as labeling. Line numbers, statement numbers and instruction addresses are the common labels that can be used.

In a given JBC program each instruction is located in a unique position of the method’s byte-code array. Due to this fact that the CFG of a method must contain all possible control points, instruction address is a proper candidate as the label. Figure 2.4 illustrates a set of instructions in a JBC program labeled with instruction byte-code address. Based on the structure of a method body code in JBC programs, which already defined, we can define the syntax of the statements in a JBC method as follows:

\[
S ::= \ell : \text{inst} ; S \mid \epsilon \quad \ell \in \text{Addr}, \text{inst} \in \text{Inst}
\]

We show the instruction set of a given method \(M\), as a set of label and instruction pairs, i.e. \(\text{code}(M) = \{(\ell_0, i_0); (\ell_1, i_1); \cdots; (\ell_n, i_n)\}\). The instruction set is ordered by the instruction’s label and there exists only one instruction with label \(\ell = 0\) as the minimum element which will be the first instruction for execution and entry point.
of the method. Again since we will use \textit{succ p} to have \textit{next instruction} address, having \( \ell_k \) as a program pointer or instruction index will not affect our definitions.

### 3.3.2 CFG Nodes

Nodes in a method control flow graph build a map of the method’s execution states. The mapping must cover all possible JVM configuration with respect to the method execution. Moreover, based on the compositional verification requirements we need to distinguish between method’s entry point, return nodes, exceptional and normal internal nodes. Therefore we need to have a formal definition of the node structure.

In our CFG configuration set we will need a special node to show the \textit{abort} state of the program which is a state that an exception is occurred in one of the methods and JVM’s attempt to find a proper handler was unsuccessful. So we extend the set of addresses by adding symbol \( \flat \) (read as flat) to the set of acceptable addresses for node building function. All nodes are tagged with pair of address and method signature which makes the node to be unique in programs address space. If a node has the minimum address, i.e. 0, then we tag it with \( e \) to show it as the entry node of the CFG. If the incoming edge is labeled with one of the return operations, the node is tagged with \( r \) and finally to show an exceptional configuration we tag the corresponding node with exception type. We define the node formally as follows.

**Definition 9 (Node)** Let \( V \) be set of the nodes and \( \text{ADDR}_0 = \text{ADDR} \cup \{ \flat \} \) as our extended address space. A node of CFG for a given method \( M \in \text{METH} \) is defined as a triple \( (M, \ell, \mathcal{T} \cup \mathcal{T}_r \cup \mathcal{T}_e) \) in which

- if \( \ell = 0 \) then \( T_e = \{ e \} \) else \( T_e = \emptyset \)
- if \( M[\ell] \in \text{RetInst} \) then \( T_r = \{ r \} \) else \( T_r = \emptyset \)

Two nodes are equal if both specify same address of the same method with equal atomic proposition sets. Meaning:

\[
(M, \ell, T) = (M', \ell', T') \iff (M = M' \land \ell = \ell' \land T = T')
\]

To have a simplified notation of nodes here we define equivalent notations for normal control nodes which is specified by \( (M, \ell, \{ \} \cup T_r \cup T_e) \) and exceptional control nodes which is defined by \( (M, \ell, x \cup T_r \cup T_e) \), where \( x \) denotes set of exception labels or other auxiliary label needs to be listed as node label.

We will use symbol \( \circ_{M}^{\ell} \) to denote a normal control node which specifies corresponding node of \( M[\ell] \). Also symbol \( \bullet_{M}^{\ell,x} \) will indicate exceptional control node of \( M[\ell] \) containing exception \( x \in \text{EXCP} \). So formally we have:

\[
\circ_{M}^{\ell} \equiv (M, \ell, \{ \} \cup T_r \cup T_e) \quad \bullet_{M}^{\ell,x} \equiv (M, \ell, x \cup T_r \cup T_e)
\]
If \( v \models x \), meaning that node \( v \) is tagged with exception \( x \) then it is called exceptional control point of graph, otherwise if \( v \not\models x \) then we just call it normal control point of graph.

### 3.3.3 CFG Extraction Rules

Now we are ready to define the rules for control flow graph construction for a given method \( M \) of a program. Previously we modeled a program as an environment \( \Gamma \) such that \( \Gamma[M] = \langle \text{code}, \text{handlers} \rangle \) denotes an implementation for a method with reference \( M \) (method signature). We have the code as an ordered set of instructions labeled with their addresses and handlers denotes the exception handling table defined for method. Actually the control flow graph rules for a method \( M \) in environment \( \Gamma \) uses the implementation of the method \( \Gamma[M] \) and build a set of the labeled edges. So CFG construction rules must be defined syntactically according to the statements appearing in method code.

Using case analysis, for each JBC instruction, possible execution paths to the next instructions are established through edges. Constructing each edge the source and target nodes need to be constructed according to the instruction type and its label. So if we define possible edges for each instruction, the nodes are implicitly constructed. In following we formally define control flow graph constructor function which given a method signature determines a set of edges. The construction rules are defined purely statical based on the method’s instructions but they intuitively justify the instruction’s operational semantic. In other words the set of labeled instructions is the only input that is given to the construction rules. The way that the rules establish the edges conform the operational semantic of the instruction.

**Definition 10 (Method Control Flow Graph)** Let \( \mathcal{V} \) be set of nodes and \( I_g = \text{Inst}_g \cup \{\text{handle}\} \) and \( \text{Inst}_g \) be any mapping from \( \text{Inst} \) to show corresponding instruction. Also let \( \Pi \) be set of programs (environments) and \( \text{MethSig} \) be set of methods signatures. Then control flow graph of a method is defined as a function: \( m\mathcal{G} : \Pi \times \text{MethSig} \to \mathcal{P}(\mathcal{V} \times \text{Inst}_g \times \mathcal{V}) \) with following construction rules.

In following section \( \Gamma \) is assumed as our default environment, so we will use \( m\mathcal{G}(M) \equiv m\mathcal{G}(\Gamma,M) \).

**Composition**

Control flow graph construction rules need to have a rule which decomposes a sequence of instructions to build the elements from individual instruction. So the control flow graph of two subsequence of JBC instructions will be the union of control flow graphs of each subsequence independently and is defined formally as follows:

\[
m\mathcal{G}(S_1;S_2,H) = m\mathcal{G}(S_1,H) \cup m\mathcal{G}(S_2,H)
\]
3.3. CFG CONSTRUCTION

Normal Computation

In this case the rules for all instructions \( i \in \text{CmpInst} \) are defined. Operational semantics rule [Cmp] shows that the control is transferred to the next instruction execution:

\[
c_v = ((M, p, f, s, z).A; h) \xrightarrow{i} ((M, \text{succ } p, f', s', z').A; h') = c'_v \quad \forall i \in \text{CmpInst}
\]

So there is a need for an edge that connects the control node, expressing the execution configuration of current instruction, to the next control node, indicating possible configuration of execution for next instruction. Using node construction to build the nodes we write the rule as follows:

\[
mG((p, i), H) = \{(o_M^p, i_g, o_M^{\text{succ } p})\} \quad \forall i \in \text{CmpInst}
\]

in which \( i_g \) can be any mapping from \( i \in \text{CmpInst} \) to show related transition instruction inside the method.

Normal Conditional

From the operational semantics [Cnd\text{tt}] and [Cnd\text{ff}] there are two possible executions for these instructions, which are next instruction if the condition is false and instruction specified at branch position if the condition is true:

\[
c_v = ((M, p, f, s, z).A; h) \xrightarrow{i} ((M, \text{succ } p, f, s', z').A; h) = c'_v \quad \forall i \in \text{CndInst}
\]

\[
c_v = ((M, p, f, s, z).A; h) \xrightarrow{i} ((M, q, f, s', z).A; h) = c'_v \quad \forall i \in \text{CndInst}
\]

So there is a need for two edges that connect the current control node to the possible next control nodes. The rule to establish these connections is defined as follows:

\[
mG((p, i), H) = \{(o_M^p, i_g, o_M^{\text{succ } p}), (o_M^p, i_g, o_M^q)\} \quad \forall i \in \text{CndInst}
\]

in which \( i_g \) can be any mapping from \( i \in \text{CmpInst} \) to show related transition instruction inside the method and \( q \) is the branch address.

Unconditional Jumps

Based on the operational semantics of [Jmp] there is only one possible execution for these instructions, which is the instruction specified at unconditional branch position:

\[
c_v = ((M, p, f, s, z).A; h) \xrightarrow{i} ((M, q, f, s, z).A; h) = c'_v \quad \forall i \in \text{JmpInst}
\]
So there is a need for an edge that connects the current control node to the unconditionally jump control node. We write the rule as follows:

\[ m_G((p, i), H) = \{ (\circ^p_M, i_g, \circ^q_M) \} \quad \forall \ i \in \text{JmpInst} \]

in which \( i_g \) can be any mapping from \( i \in \text{CmpInst} \) to show related transition instruction inside the method and \( q \) is the jump address.

**Exceptions**

In this case the rules for instructions that their executions might cause exceptions are built. Rules \([\text{Xmp}_1], [\text{Xmp}_1], [\text{Exc}_1], [\text{Exc}_2] \) and \([\text{Exc}_3] \) show that there are two possible executions: normal and exceptional. In normal case the execution control is transferred to the next instruction but in exceptional case an exceptional record is pushed on the stack and JVM takes the execution control. Then if the correct handler is found, the execution control is passed to the handler. Otherwise activation record related to the method being executed is removed from stack and caller method takes the execution control. Finally if there was not any proper handler in all callers sequence in activation records stack, the program abnormally terminates in an exceptional state which is called *abort*. Following shows the all possible state transitions:

\[
\begin{align*}
c_v = ((M, p, f, s, z).A; h) & \xrightarrow{\sim} ((M, succ\ p, f', s', z').A; h') = c'_{v} \\
c_v = ((M, p, f, s, z).A; h) & \xrightarrow{(\chi)_{\text{exc}}} ((x)_{\text{exc}}.M, p, f, s, z).A; h') = c'_{v} \\
c_v = (x)_{\text{exc}}.M, p, f, s, z).A; h) & \xrightarrow{(x)_{\text{exc}}} (M, t, f, s, z).A; h') = c'_{v} \\
c_v = (x)_{\text{exc}}.M, p, f, s, z).A; h) & \xrightarrow{(x)_{\text{exc}}} (x)_{\text{exc}}.A; h') = c'_{v} \\
c_v = (x)_{\text{exc}}.M, p, f, s, z).A; h) & \xrightarrow{(x)_{\text{exc}}} (x)_{\text{exc}}.A; h') = c'_{v} \\
\end{align*}
\]

So we need to add all possible edges which are: successful execution, exceptional execution, successful handling and failed handling. Here we need a function that given an instruction \( i \in \text{XmpInst} \) defines set of exceptions that execution of \( i \) with unsatisfied conditions can cause. We introduce function \( \chi : \text{XmpInst} \rightarrow 2^{\text{Excp}} \) to find all possible exceptions of a given instruction and is defined as follows for two sample instructions. The others can be defined as well:

\[
\chi(i) = \begin{cases} 
\{ \text{NullPointerExceptionExc}, \text{OutOfIndexExc} \} & \text{if } i = \text{arrayload} \\
\{ \text{ArithmaticExc} \} & \text{if } i = \text{div} 
\end{cases}
\]

Now using this function, we define the rule as the union of a singleton set containing the edge for normal execution and the set of all possible exceptional edges. The rule is defined as follows:

\[ m_G((p, i), H) = \{ (\circ^p_M, i_g, \circ^q_M) \} \cup \mathcal{E}_g^i \quad \forall \ i \in \text{XmpInst} \]
3.3. CFG CONSTRUCTION

where

\[ E_i^g = \bigcup \left\{ \{(\circ_M^{p_i}, \iota_{g_i}, \bullet_M^{p_i(x)}), \gamma \big| \gamma \in \chi(i)\} \cup \mathcal{H}_g^x \right\} \]

in which set \( \mathcal{H}_g^x \) with following definition is used to denote required edges for exception handling.

\[ \left( \Gamma \vdash x <; y \right) \implies \]

\[
\mathcal{H}_g^x = \left\{ \begin{array}{ll}
\{(p_M^{x}, \text{handle}, c_M^i)\} & h_{\Gamma[M]}(p, y) = t \neq 0 \\
\{(p_M^{x}, \text{handle}, \bullet_M^{x,r})\} & h_{\Gamma[M]}(p, y) = 0 \land M \notin M_e(Prg) \\
\{(p_M^{x}, \text{handle}, \circ_M^i)\} & h_{\Gamma[M]}(p, y) = 0 \land M \in M_e(Prg)
\end{array} \right. 
\]

in which \( M_e(Prg) \) is the set of methods of program \( Prg \) which are the entries of an execution.

The definition simply makes an edge to the abort node \( \bullet_M^{x,r} \) if the method is an execution entry. The node is marked with exception type \( x \) to show the type of the exception that forced the program to terminate abnormally. If the method does not belong to the program entry methods set, then an exceptional edge to \( \bullet_M^{x,r} \) is added. Since there is no proper handler for thrown exception and current activation record (method \( M \)) will be removed from the stack (return from method), exceptional node is labeled as return node. In case of finding a proper handler for the exception, an edge to \( \circ_M^i \) is established and configuration returns to its normal situation.

**Throw**

In this case we build the rules for instructions that their executions explicitly push exception on the stack. Operational semantics rules \([\text{Thr}]\) clarifies that after the execution an exceptional record is pushed on the stack and the control of execution is taken by JVM to handle exception. Then like instructions in XMPINST, if the correct handler is found, the control is returned back to the handler otherwise activation record related to the method being executed is removed from the stack (return from method), exceptional node is labeled as return node. In case of finding a proper handler for the exception, an edge to \( \circ_M^i \) is established and configuration returns to its normal situation.

Following shows the all possible state transitions:

\[
c_v = ((M, p, f, s, z).A; h) \xrightarrow{\text{exc}} ((x)_{\text{exc}}.\langle M, p, f', s', z\rangle.A; h') = c'_v
\]

\[
c_v = ((x)_{\text{exc}}.\langle N, q, f, s, z\rangle.\langle M, p, f', s', z\rangle.A; h) \xrightarrow{\text{exc}} ((x)_{\text{exc}}.\langle M, p, f', s', z\rangle.A; h') = c'_v
\]

37
So we need to construct all possible edges which are: all exceptional executions and either successful or failed handling. Like handling edges set defined in exceptions case, using $\mathcal{H}_g^e$ the rule is defined as follows:

$$mG((p,i), H) = \{(\epsilon^p_M, i_g, \epsilon^p_M)\} \cup \mathcal{H}_g^e \quad \forall \; i \in \text{ThrInst}$$

where

$$\begin{align*}
(i = \text{throw } & \varrho \land \Gamma \vdash \varrho < y) \implies \\
\mathcal{H}_g^e = & \begin{cases} 
\{(\epsilon^p_M, \text{handle}, \epsilon^t_M)\} & h_{\Gamma[M]}(p, y) = t \neq 0 \\
\{(\epsilon^p_M, \text{handle}, \epsilon^v_M)\} & h_{\Gamma[M]}(p, y) = 0 \land M \notin M_e(Prg) \\
\{(\epsilon^p_M, \text{handle}, \epsilon^r_M)\} & h_{\Gamma[M]}(p, y) = 0 \land M \in M_e(Prg)
\end{cases}
\end{align*}$$

To show the exception in throw instruction, $\varrho$ is used as a known exception, because the type is identifiable from the instruction. Like exceptions case $\mathcal{H}_g^e$ is used to make handling edges for $\varrho$.

**Invocations**

This case deals with defining rules for instructions that are of type procedure call. Their execution cause execution control to be transferred to the target method. Operational semantics rules [INV] and [XNV] show that there are possible normal and exceptional executions. In case of normal execution the activation record of target method is pushed on the stack, which will be the execution context. If the object being called be null then an exception of type $\varrho_N$=NullPointerException, is pushed on the stack and proceeding transitions are performed by [Exc_k] rules.

What makes the invocations case complex is finding the possible targets of message receivers and handling uncaught exceptions in callee. If the instruction is invokevirtual or invokevirtual then there will just one possible receiver which is the static type of the object. But if the instruction is invokevirtual then the set of possible receivers is determined using static analysis algorithms for virtual method call resolution which has been explained previously.

If the target method’s execution terminates normally then the control will be returned back to the caller and the next instruction of the caller will be fetched. But if there is a failed handling exception in the callee then the rule [Exc_c2] shows that the caller will be responsible for handling this exception. So CFG construction rules for invocations must define the required edges for both $\varrho_N$ and all exceptions like $x$ which are the result of target method execution. Recalling that $m, n$ stand for
methods’ simple name and $M,N$ stand for methods’ signature (including package, class names, simple name and type), the rule for invocation is defined as follows:

$$mG((p,i),H) = \{(o^p_M,iq,\bullet^p_M)\} \cup R^i_g \cup H^\text{en}_g \cup N^x_g \quad \forall \ i \in \text{InvInst}$$

where $R^i_g$ indicates the set of edges with possible receivers’ call label, $H^\text{en}_g$ is the set of edges to handle $\rho_N=\text{NullPointerException}$ and $N^x_g$ defines set of edges to handle all uncaught exceptions from possible callees. Following defines the set $R^i_g$:

$$R^i_g = \{(\circ^p_M,\text{call} (\tau,N),\circ^\text{nullp}_M) \mid \tau \in \text{Rec}_\Gamma(i)\}$$

Function $\text{Rec}_\Gamma(i)$ determines the set of possible receivers of the method call in environment $\Gamma$. If the instruction is invokevirtual($o$,N) then it will use virtual method call resolution function $\text{res}^\alpha$. It can be defined as follows:

$$\text{Rec}_\Gamma(i) = \begin{cases} \{\text{static}T(o)\} & \text{if } i \in \{ \text{invokespecial } (o,N), \text{invokestatic } (o,N)\} \\ \text{res}^\alpha & \text{if } i = \text{invokevirtual } (o,N) \end{cases}$$

To have a review, if we assume that virtual method call resolution is done by the algorithm $\alpha = \text{RTA}$ then the result of the resolution for object $o$ and method $N$ in environment $\Gamma$ will be:

$$\text{res}^\alpha(o,N) = \{\tau \mid \tau \in I\text{C}_\Gamma \land \Gamma \vdash \tau <: \text{static}T(o) \land \text{sig}(n) = \text{lookup}(n,\tau)\}$$

in which $I\text{C}_\Gamma$ is for set of instantiated classes in environment $\Gamma$, $\text{static}T(o)$ gives the static type of object $o$ and $\text{lookup}(n,\tau)$ specifies the signature of method named $n$ in class $\tau$.

Since there is a possibility that target object’s reference be null, which causes null pointer exception for invocation instruction, we need to have a set of handlers for this specific type of instruction. $H^\text{en}_g$ is defined as following, makes set of handlers edges for the rule:

$$\Gamma \vdash \rho_N <: y \implies$$

$$\begin{align*}
\text{H}^\text{en}_g = & \begin{cases} 
\{(\bullet^p_M,\text{handle},\circ^p_M)\} & h_{\Gamma[M]}(p,y) = t \neq 0 \\
\{(\bullet^p_M,\text{handle},\bullet^p_M)\} & h_{\Gamma[M]}(p,y) = 0 \land M \notin M_e(Pr_g) \\
\end{cases} \\
& \begin{cases} 
\{(\bullet^p_M,\text{handle},\bullet^p_M)\} & h_{\Gamma[M]}(p,y) = 0 \land M \in M_e(Pr_g) \\
\end{cases}
\end{align*}$$

39
The construction rule must also specify set of handling edge s for exceptions which are not caught in possible callees \((N_k)\) and exceptional returns from them forces JVM to look for handlers in current active method \((M)\).

We remember that \(\text{res}_\alpha^\Gamma\) resolves the set of possible types like \(T\) containing method \(N\) for \(\text{invokevirtual}(0,N)\). Now we need to find exceptional return nodes from all possible receivers. So possible edges for uncaught exceptions in receivers resolved by \(\text{res}_\alpha^\Gamma(o,N)\) is defined as follows:

\[
N^x_g = \bigcup \left\{ \left\{ (c^p_M, \text{handle}, a^p_M, x) \right\} \cup H^tx_g \mid a^p_{N_k} \in mG(N_k), N_k \in \text{res}_\alpha^\Gamma(o,N) \right\}
\]

where

\[
\Gamma \vdash x : y \implies
\]

\[
H^tx_g = \begin{cases}
\left\{ (a^p_M, \text{handle}, c^p_M) \right\} & h_{\Gamma[M]}(p,y) = t \neq 0 \\
\left\{ (a^p_M, \text{handle}, a^p_M, x, r) \right\} & h_{\Gamma[M]}(p,y) = 0 \land M \notin M_e(Pr_g) \\
\left\{ (a^p_M, \text{abort}, a^p_M, x, r) \right\} & h_{\Gamma[M]}(p,y) = 0 \land M \in M_e(Pr_g)
\end{cases}
\]

Like handling edges defined for exceptional instructions, \(H^tx_g\) adds all possible exceptional edges for successful/failed handlers for all exceptional nodes in \(CFG_{N_k}\) which is the control flow graph of method \(N_k \in \text{res}_\alpha^\Gamma(o,N)\).

Returns

Return instructions are the last instructions for normal execution in the method instruction set. As it is seen from node definition function, if a node is of type return instruction, it is recognized and marked as return node. So when the control flow graph construction rule finds any \(i \in \text{RETINST}\) then it does not need to make any element, except when there is only one instruction of type return in the method instruction set. The rule is defined formally as:

\[
mG((p,i),H) = \begin{cases}
0, \{r\} & \text{if } p = 0 \\
\emptyset & \text{otherwise}
\end{cases}
\]

Figure 3.3 collects all the CFG construction rules. All in all, the control flow graph construction function, given an instruction sequence of \(\Gamma[M]\), tries to decompose the sequence to reach individual instructions, and using construction rules for each instruction, builds up the complete control flow graph of the method.
3.3. CFG CONSTRUCTION

Before we start to define the behavior of CFG and prove the correctness of extracted CFG we need to explain extracted flow graphs in terms of the program’s model which is being used in compositional verification tool set [2] (definitions 2 and 5). Gurov et al. represent programs as collections of methods control-flow graphs equipped with interfaces of provided and required methods. In this representation the program composition forms the disjoint union of the respective collections of method graphs and allows the composed program to communicate via method invocation [1].

In the flow graph construction rules (definition 10), we saw that the extracted graph for a given method represents a complete control points of the method’s execution. It contains nodes for both normal and exceptional control points. Method specification which is equipped with exceptions in definition 5, assumes that each control node cannot be tagged with more than one exception. This results to have a complete set of possible execution paths. Each thrown exception leads to a different execution path. Some might be managed by a proper handler but probably there are still some exceptional control nodes leading to an exceptional return.

3.3.4 CFG as a Transition System

In the flow graph construction rules (definition 10), we saw that the extracted graph for a given method represents a complete control points of the method’s execution. It contains nodes for both normal and exceptional control points. Method specification which is equipped with exceptions in definition 5, assumes that each control node cannot be tagged with more than one exception. This results to have a complete set of possible execution paths. Each thrown exception leads to a different execution path. Some might be managed by a proper handler but probably there are still some exceptional control nodes leading to an exceptional return.

Figure 3.3. CFG Construction Rules
CHAPTER 3. FLOW GRAPH IN JBC: FORMAL STUDY

We still keep the condition and redefine the method specification with exceptions (definition 5) induced by our defined flow graph extraction function $mG$ (definition 10).

Definition 11 (Method Specification with Exceptions) A method flow graph with exceptions for $M \in \text{MethSig}$ and Excp as set of exceptions, is a finite model $\mathcal{M}_M = (V_M, L_M, \rightarrow_M, A_M, \lambda_M)$, where

- $V_M$ is the set of control nodes,
- $L_M = \text{Inst}_g \cup \{\text{handle}\}$
- $A_M = \{M, e, r\} \cup \text{Excp}$, and $M \in \lambda_M(v)$ for all $v \in V_M$, and for all $x, x' \in \text{Excp}$ if $\{x, x'\} \subseteq \lambda_M(v)$ then $x = x'$, i.e., each control node is tagged with at most one exception.
- $\rightarrow_M \subseteq V_M \times L_M \times V_M$

A method specification for $M \in \text{MethSig}$ is a specification $(\mathcal{M}_M, E_M)$ such that $\mathcal{M}_M = mG(M)$ is a method graph with exceptions for $M$ and $E_M \subseteq V_M$ a non-empty set of entry points of $M$, and for all $v \in E_M$, $e \in \lambda(v)$.

Example 4 (Method Specification with Exceptions) Figure 3.4 shows a call to method even of class EvenOdd, which implements methods illustrated in figure 1.3. The only difference here is that methods of EvenOdd are not static and object needs to be instantiated. CFG of the method shows the elements. We remove method names from nodes and labels from edges for simplicity.

The program model presented in [2] introduces two constraints for method specifications with exceptions which are called well-formedness constraints:

1. Entry nodes are not exceptional
2. All outgoing edges from exceptional control points are internal transfer edges ending in a normal control point

Satisfaction of second constraint results to have a clean behavior of the graph in which catching an exception always results in a normal state in the same method. But it is considered as an optional constraint.

Based on CFG extraction rules defined, the second constraint is not satisfied. In $mG(M)$ if there is no proper handler for an exceptional node the outgoing edge might lead to an exceptional return control node. So we are not able to assume all method specifications to be well-formed. Only if all exceptions of a particular method have correct handler then its specification will be well-formed. So we need to extend our flow graph behavior in case of exceptional returns.

The following defines the behavior of a CFG as a labeled transition system in which we extend the CFG behavior definition from [13] with rule $[x\text{return}]$. In
Code:
0: aconst_null
1: astore_1
2: aload_1
3: bipush 10
5: invokevirtual #2; // Method EvenOdd.even:(I)Z
8: pop
9: goto 28
12: astore_2
13: new #4; // class EvenOdd
16: dup
17: invokevirtual #5; // Method EvenOdd."<init>":()V
20: astore_1
21: aload_1
22: bipush 10
24: invokevirtual #2; // Method EvenOdd.even:(I)Z
27: pop
28: return

Exception table:
from   to target type
2     9    12 Class java/lang/NullPointerException

Figure 3.4. Specification with exceptions

this definition $M_1$ call $M_2$ denotes an invocation from $M_1$ to $M_2$ when method $M_2$ is provided and the constructed edge between two control nodes is labeled with call $M_2$. $M_1$ ret $M_2$ and $M_1$ xret $M_2$ show normal and exceptional returns from a method call. If an exception is thrown in a given control node, the next control node will be tagged with the exception type and throw $x$ shows the transition in which $x$ indicates the exception. We use catch $x$ for the transition in exception handling edges. Label $\tau$ is used for all internal normal transitions. Definition 12 uses the following abbreviation: $v \models \text{Excp} \iff \exists x \in \text{Excp}. v \models x$

Definition 12 (CFG Behavior) Let $G = (M, E) : I$ be a closed flow graph with exceptions such that $M = (V, L, \rightarrow, A, \lambda)$. The behavior of $G$ is described by the specification $b(G)$, where $M_g = (S_g, L_g, \rightarrow_g, A_g, \lambda_g)$ such that:

- $S_g \in V \times (V)^*$, i.e., states are pairs of control nodes and stacks of control nodes,
- $L_g = \{\tau\} \cup L^C_g \cup L^X_g$ where
  - $L^C_g = \{ M_1 \rightarrow \leftarrow M_2 \mid l \in \{\text{call, ret, xret}\}, M_1, M_2 \in I^+\}$ denotes set of call labels and
  - $L^X_g = \{ l \ x \mid l \in \{\text{throw}, \text{catch}\}, x \in \text{Excp}\}$ is for set of exceptional transitions labels.
• $A_g = A$
• $\lambda_g((v,\sigma)) = \lambda(v)$
• $\rightarrow_g \subset S_g \times S_g$ is the set of transitions in $CFG_M$ with following rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[transfer]</td>
<td>$(v,\sigma) \xrightarrow{\tau} g (v',\sigma)$</td>
<td>if $M \in I^+, v \xrightarrow{i_\tau} M v'$ $v \models \neg r, v' \not\models \text{EXCP}$</td>
</tr>
<tr>
<td>[call]</td>
<td>$(v_1,\sigma) \xrightarrow{M_1 \text{ call } M_2} g (v_2, v_1.\sigma)$</td>
<td>if $M_1,M_2 \in I^+, v_1 \xrightarrow{\text{call } M_2} M_1 v'_1$, $v'_1 \in \text{next}(v_1), v_1 \not\models \text{EXCP}$ $v_2 \models M_2, v_2 \in E, v_1 \models \neg r$</td>
</tr>
<tr>
<td>[return]</td>
<td>$(v_2, v_1.\sigma) \xrightarrow{M_2 \text{ ret } M_1} g (v'_1, \sigma)$</td>
<td>if $M_1,M_2 \in I^+, v_2 \models M_2 \wedge r$ $v_1 \models M_1, v'_1 \not\models \text{EXCP}, v'_1 \in \text{next}(v_1)$</td>
</tr>
<tr>
<td>[xreturn]</td>
<td>$(v_2, v_1.\sigma) \xrightarrow{M_2 \text{ xret } M_1} g (v'_1, \sigma)$</td>
<td>if $M_1,M_2 \in I^+, v_2 \models M_2, v_1 \models M_1$ $v_2 \xrightarrow{\text{handle}} M_2 v'_2, v_1 \xrightarrow{\text{handle}} M_1 v'_1$ $v_2 \models X, v'_2 \models X \wedge r, v_1 \not\models X, v'_1 \models X$ $X \in \text{EXCP}$</td>
</tr>
<tr>
<td>[throw]</td>
<td>$(v,\sigma) \xrightarrow{\text{throw } x} g (v',\sigma)$</td>
<td>if $M \in I^+, v \xrightarrow{i_{\text{throw } x}} M v'$ $v \models \neg r, v' \models \text{EXCP}$</td>
</tr>
<tr>
<td>[catch]</td>
<td>$(v,\sigma) \xrightarrow{\text{catch } x} g (v',\sigma)$</td>
<td>if $M \in I^+, v \xrightarrow{\text{handle } M} v'$ $v \models \neg r, v \models \text{EXCP}, v' \models \neg r, v' \not\models \text{EXCP}$</td>
</tr>
</tbody>
</table>

**Example 5 (CFG Behavior)** Figure 3.5 shows two CFG of methods $M$ and $N$. Nodes are marked with indexes (they are not instruction addresses). Following illustrates some normal and exceptional behaviors:

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{M \text{ call } N} (v_3, \varepsilon) \xrightarrow{\tau} (v_4, \varepsilon) \xrightarrow{N \text{ ret } M} (v_5, \varepsilon) \xrightarrow{\tau} (v_6, \varepsilon) \xrightarrow{\tau} (v_7, \varepsilon) \ldots$$

$$(v_{13}, v_6) \xrightarrow{\tau} (v_{14}, v_6) \xrightarrow{\text{throw } X_1} (v_{15}, v_6) \xrightarrow{N \text{ xret } M} (v_{16}, v_6) \xrightarrow{\tau} (v_{17}, v_6) \xrightarrow{\tau} (v_{18}, v_6) \xrightarrow{\tau} \ldots$$

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\tau} (v_3, \varepsilon) \xrightarrow{\text{throw } X_2} (v_{19}, v_6) \xrightarrow{N \text{ xret } M} (v_{20}, v_6) \xrightarrow{\tau} (v_{21}, v_6) \xrightarrow{\tau} \ldots$$

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\tau} (v_3, \varepsilon) \xrightarrow{\text{catch } X_2} (v_4, \varepsilon) \xrightarrow{\tau} (v_5, \varepsilon) \xrightarrow{M \text{ ret } \ldots} \ldots$$

44
3.4 CFG Correctness

In this section the correctness of the constructed control flow graph of a given method is proved. We prove that extracted control flow graph is sound, meaning that $\text{CFG}$ is able to match all possible moves that method execution may produce.

To continue, first, based on the CFG node definition we define a mapping from JVM configurations to CFG behavioral configurations (states). We need to abstract JBC execution state space by a transformation function $\theta$. Then based on this transformation function, we prove that behavior of a CFG simulates behavior of the corresponding method in JBC.

**Definition 13 (Configuration Transformation)** Let $V_{MC}$ be the set of JVM execution configurations and $S_g$ set of states in $M_g$, i.e. set of the states in constructed flow graph of method $M$. Then $\theta : V_{MC} \rightarrow S_g$ is defined inductively as follows:

1. $\theta((M,p,f,s,z).A;h) = (p_M^p,\theta(A;h))$
2. $\theta((M,p,f,s,z).\epsilon;h) = (p_M^\epsilon)$
3. $\theta((x)_{exc}.\epsilon;h) = (b_M^{x,r},\epsilon)$
4. $\theta((x)_{exc}.(M,p,f,s,z).A;h) = (p_M^{\{x\},\theta(A;h)})$

---

**Figure 3.5.** Specification with exceptions
The CFG correctness is proved in terms of simulation relation which is defined here along with an example.

**Definition 14 (Simulation)** A simulation is a binary relation \( R \) on \( S \) such that whenever \((s, t) \in R\) then \( \lambda(s) = \lambda(t) \), and whenever \( s \xrightarrow{a} s' \) then there is some \( t' \in S \) such that \( t \xrightarrow{a} t' \) and \((s', t') \in R\). We say that \( t \) simulates \( s \), written \( s \preceq t \), if there is a simulation \( R \) such that \((s, t) \in R\).

**Example 6 (Simulation)** Figure 3.6 illustrates a simulation \( R \) on set of states \( S = \{v_0, \cdots, v_{10}\} \). We can see that \((v_0, v_5) \in R\) because \( \lambda(v_0) = \lambda(v_5) \), \( v_0 \xrightarrow{\varepsilon} v_1 \), \( v_5 \xrightarrow{\varepsilon} v_6 \) and \((v_1, v_6) \in R\) because \( \lambda(v_1) = \lambda(v_6) \) and if \( v_1 \xrightarrow{\varepsilon} v_2 \) then \( v_6 \xrightarrow{\varepsilon} v_7 \) and \((v_2, v_7) \in R\) because \( \lambda(v_2) = \lambda(v_7) \), \( v_2 \xrightarrow{\varepsilon} v_3 \), \( v_7 \xrightarrow{\varepsilon} v_9 \) and \((v_3, v_9) \in R\) because \( \lambda(v_3) = \lambda(v_9) \) and there is no more transitions outgoing from \( v_3 \); and if \( v_1 \xrightarrow{\varepsilon} v_4 \) then \( v_6 \xrightarrow{\varepsilon} v_8 \) and \((v_4, v_8) \in R\) because \( \lambda(v_4) = \lambda(v_8) \) and there is no more transitions outgoing from \( v_4 \).

In fact \( M' \) can follow all moves in transition machine named as \( M \) starting from \( v_0 \). But \( M \) is not able to follow all moves of \( M' \).

![Figure 3.6. Example: Simulation Relation \((v_0, v_5) \in R\)](image)

In order to prove the correctness of the constructed CFG, for a given shape of a JVM configuration and instruction being executed we deduce the possible transitions from the instruction operational semantics. Then we deduce the matching CFG configuration by the configuration transformation function \( \theta \). Afterwards based on the CFG construction rules we show the possible established edges. These possible edges give the possible transitions paths to the next CFG configuration states. The next possible CFG configuration must be an image of the next possible JVM configuration states based on the function \( \theta \). If for all possible JVM configuration states and all possible instruction groups we can find constructed CFG edges and CFG configurations, then we prove that control flow graph extraction rules construct all possible execution paths of the program. We need to note here that we use simulation modulo relabeling by mapping the transition labels in JVM, i.e. \( \text{Inst} \cup \{\varepsilon\} \), to the transition labels in CFG, i.e. \( \text{Inst}_g \cup \{\text{handle}\} \).
3.4. CFG CORRECTNESS

**Theorem 1 (CFG Simulation)** For a closed program $\mathcal{P}$ and corresponding flow graph $\mathcal{G}$ behavior of $\mathcal{G}$ simulates the execution of $\mathcal{P}$.

**Proof:** The proof is done using case analysis on $\text{VMC}$ as follows:

**Case** $c_v = (\langle M, 0, f, \epsilon, z \rangle.A; h)$ This case is for entry point execution of a method $M$ in $\mathcal{P}$. There are two possible cases for entry states:

1. In first case a concrete method has return type `void` and there is no implementation. So the instruction set of the method has a single member:

   \[ M[!] = \langle (0, \text{return}) \rangle \]

   Based on the transformation function definition we have:

   \[ \theta(\langle M, 0, f, \epsilon, z \rangle.A; h) = \langle \circ_0^M, \theta(A; h) \rangle \]

   On the other hand because of the CFG construction rule for $i \in \text{RetInst}$ and node definition we can guarantee that

   \[ \circ_0^M \in mG(M) \]

   And based on the definition of entry nodes in a method’s control flow graph and node definition we can indicate $\circ_0^M$ as entry node of $\text{CFG}_M$. So we have:

   \[ \circ_0^M \in E_M \]

   Therefore,

   \[ \theta(c_v) = (\circ_0^M, \sigma) = s_g \]

   In this case since there is no successor states defined for method execution, there will not be any successor nodes for $\circ_0^M$, i.e. $(\text{next}(\circ_0^M) = \emptyset)$ in CFG, as well. So the proof terminates here and

   \[ \therefore \quad \theta(c_v) = s_g \]

2. In second case the concrete method contains implementation. So the instruction set for method body has more than one member. Based on the transformation function definition we have:

   \[ \theta(\langle M, 0, f, \epsilon, z \rangle.A; h) = \langle \circ_0^M, \theta(A; h) \rangle \]

   On the other hand because of the CFG construction rules for $i \in \text{Inst}$ and node definition we can guarantee that

   \[ \circ_0^M \in mG(M) \]

47
CHAPTER 3. FLOW GRAPH IN JBC: FORMAL STUDY

So based on the definition of entry nodes in a method’s control flow graph and node definition we can indicate \( c_0^0 \) as entry node of \( mG(M) \). So we have:

\[
\circ_0^0 \in \mathbb{E}_M
\]

Hence,

\[
\theta(c_v) = (c_0^0, \sigma) = s_g
\]

\[
\therefore \theta(c_v) = s_g
\]

Then successors of entry states will be studied to complete the proof.

We should note here that uniqueness of an instruction with \( p = 0 \) and the node definition guarantees that \( c_0 \) is unique in \( CFG \) of \( M \).

**Case** \( c_v = \langle (M, p, f, s, z) \cdot A; h \rangle \land i = M[p] \in \text{CmpInst} \): First we find result states based on the operational semantics for byte-code instructions.

\[
i \in \text{CmpInst} \land c_v = \langle (M, p, f, s, z) \cdot A; h \rangle \implies
\]

\[
\langle ((p,i);S),c_v \rangle \xrightarrow{i} \langle ((\text{succ}\ p, i');S'),c'_{v} \rangle \quad \therefore [\text{CmpInst}]
\]

where , \( c'_v = \langle (M, \text{succ}\ p, f', s', z').A; h' \rangle \)

On the other hand,

\[
\theta(c_v) = (c_{v}^p, \theta(A; h)) \implies s_g = (c_{v}^p, \sigma)
\]

Furthermore,

\[
\theta(c'_{v}) = (c_{v}^{\text{succ}\ p}, \theta(A; h')) \implies s'_{g} = (c_{v}^{\text{succ}\ p}, \sigma)
\]

According to the flow graph construction rules we have:

\[
(c_{v}^p, i_g, c_{v}^{\text{succ}\ p}) \in mG(M)
\]

Therefore for all \( i \in \text{CmpInst} \):

\[
c_v \xrightarrow{i} c'_{v} \implies (c_{v}^p \xrightarrow{i_g} c_{v}^{\text{succ}\ p}) \implies \theta(c_v) \xrightarrow{\tau} \theta(c'_{v})
\]

**Case** \( c_v = \langle (M, p, f, s, z) \cdot A; h \rangle \land i = M[p] \in \text{CndInst} \) First we find result states based on the operational semantics for byte-code instructions:

\[
i \in \text{CndInst} \land c_v = \langle (M, p, f, s, z) \cdot A; h \rangle \implies
\]

\[
\langle ((p,i);S),c_v \rangle \xrightarrow{i} \langle ((\text{succ}\ p, i');S'),c'_{v} \rangle \quad \therefore [\text{CndInst}^{\text{ff}}]
\]

\[
\lor
\]

48
3.4. CFG CORRECTNESS

\[ ((\langle p, i \rangle ; S), c_v) \xrightarrow{i} ((\langle q, i'' \rangle ; S''), c''_v) \vdash [\text{CndInst}^\text{tt}] \]

where, \n\n\[ c'_v = ((M, \text{succ } p, f, s', z). A; h) \]

and \n\n\[ c''_v = ((M, q, f, s', z). A; h) \]

On the other hand,

\[ \theta(c_v) = \langle c^p_M, \theta(A; h) \rangle \Rightarrow s_g = (c^p_M, \sigma) \]

Furthermore,

\[ \theta(c'_v) = \langle c'^{\text{succ } p}_M, \theta(A; h') \rangle \Rightarrow s'_g = (c'^{\text{succ } p}_M, \sigma) \]

and also,

\[ \theta(c''_v) = \langle c'^q_M, \theta(A; h) \rangle \Rightarrow s''_g = (c'^q_M, \sigma) \]

Based on graph construction rules:

\[ \{ (c'^p_M, i_g, c'^{\text{succ } p}_M), (c'^p_M, i_g, c'^q_M) \} \subset mG(M) \quad \forall i \in \text{CndInst} \]

Therefore for all \( i \in \text{CndInst} \):

\[ c_v \xrightarrow{i} c'_v \Rightarrow (c'^p_M \xrightarrow{\text{succ } p} M \xrightarrow{i_g} c'^{\text{succ } p}_M) \Rightarrow \theta(c_v) \xrightarrow{\tau_g} \theta(c'_v) \]

and

\[ c_v \xrightarrow{i} c''_v \Rightarrow (c'^p_M \xrightarrow{i_g} M \xrightarrow{i_g} c'^q_M) \Rightarrow \theta(c_v) \xrightarrow{\tau_g} \theta(c''_v) \]

**Case** \( c_v = ((M, p, f, s, z). A; h) \land i = M[p] \in \text{JmpInst} \): First we find result state based on the operational semantics for byte-code instructions.

\[ i \in \text{JmpInst} \land c_v = ((M, p, f, s, z). A; h) \Rightarrow \]

\[ ((\langle p, i \rangle ; S), c_v) \xrightarrow{i} ((\langle q, i' \rangle ; S'), c'_v) \vdash [\text{JmpInst}] \]

where, \n\n\[ c'_v = ((M, q, f, s', z). A; h) \]

On the other hand,

\[ \theta(c_v) = \langle c^p_M, \theta(A; h) \rangle \Rightarrow s_g = (c^p_M, \sigma) \]
Furthermore,\[\theta(c'_v) = (c^q_M, \theta(A); h') \implies s'_g = (c^q_M, \sigma)\]

Based on graph construction rules:

\[(c^p_i, \iota_g, c^q_M) \in mG_M \quad \forall i \in \text{CndInst}\]

Therefore for all \(i \in \text{JmpInst}\):

\[c_v \xrightarrow{i} c'_v \implies (c^p_M \xrightarrow{i_g} c^q_M) \implies \theta(c_v) \xrightarrow{\iota_g} \theta(c'_v)\]

**Case** \(c_v = (\langle M, p, f, s, z \rangle \cdot A; h) \land i = M[p] \in \text{ThrInst} : \) First we find result state based on the operational semantics for byte-code instructions.

\[i \in \text{ThrInst} \land c_v = (\langle M, p, f, s, z \rangle \cdot A; h) \implies (\langle (p, i) ; S \rangle, c_v) \xrightarrow{i} (\langle (p, i) ; S \rangle, c'_v) \quad \therefore [\text{ThrInst}]\]

where,

\[c'_v = (\langle g \rangle_{\text{exc}} \cdot (M, p, f, s, z) \cdot A; h)\]

On the other hand,

\[\theta(c_v) = (c^p_M, \theta(A); h) \implies s'_g = (c^p_M, \sigma)\]

Furthermore,

\[\theta(c'_v) = (\bullet^p_M \theta(A); h) \implies s'_g = (\bullet^p_M \theta, \sigma)\]

Based on graph construction rules:

\[(c^p_M, \iota_g, \bullet^p_M \{e\}) \in mG(M) \quad \forall i \in \text{ThrInst}\]

Therefore for all \(i \in \text{ThrInst}\):

\[c_v \xrightarrow{i} c'_v \implies (c^p_M \xrightarrow{i_g} \bullet^p_M \{e\}) \implies \theta(c_v) \xrightarrow{\text{throw } e} \theta(c'_v)\]

But as we know after this transition there will be an exception record on the stack, so JVM will take the control and there will be *silent transitions* to deal with this situation. So the case will be considered when

\[c_v = (\langle x \rangle_{\text{exc}} \cdot (M, p, f, s, z) \cdot A; h)\]

or

\[c_v = (\langle x \rangle_{\text{exc}} \cdot (M, p, f, s, z) \cdot \epsilon; h)\]
3.4. CFG CORRECTNESS

Case $c_v = ( ⟨M, p, f, s, z⟩.A; h) \land i = M[p] \in \text{XmpInst}$: First we find result state based on the operational semantics for byte-code instructions. There are two normal and exceptional possible executions:

\[i \in \text{XmpInst} \land c_v = ( ⟨M, p, f, s, z⟩.A; h) \implies \]

\[\langle (p,i);S\rangle, c_v \xrightarrow{i} \langle (p,i);S\rangle, c'_v \quad : [\text{Xmp1}] \]

\[\lor\]

\[\langle (p,i);S\rangle, c_v \xrightarrow{i} \langle (\text{succ}p,i');S'\rangle, c''_v \quad : [\text{Xmp2}]\]

where,

\[c'_v = ( ⟨M, \text{succ} p, f', s', z'⟩.A'; h')\]

and

\[c''_v = (⟨x⟩_{\text{exc}}.⟨M, p, f, s', z⟩.A; h)\]

On the other hand,

\[\theta(c_v) = ⟨\circ_M^p, \theta(A; h)⟩ \implies s_g = (\circ_M^p, \sigma)\]

Furthermore,

\[\theta(c'_v) = ⟨\circ_M^{\text{succ} p}, \theta(A; h)⟩ \implies s'_g = (\circ_M^{\text{succ} p}, \sigma)\]

and

\[\theta(c''_v) = ⟨\circ_M^{\{x\}}, \theta(A; h)⟩ \implies s''_g = (\circ_M^{\{x\}}, \sigma)\]

Based on graph construction rules:

\[⟨\circ_M^p, i_g, \circ_M^{\text{succ} p}⟩ \in m\mathcal{G}(M) \forall i \in \text{XmpInst}\]

also

\[⟨\circ_M^p, i_g, \circ_M^{\{x\}}⟩ \in m\mathcal{G}(M) \forall i \in \text{XmpInst}\]

Therefore for all $i \in \text{XmpInst}$:

\[c_v \xrightarrow{i} c'_v \implies (\circ_M^p \circ_M^{\text{succ} p}) \implies \theta(c_v) \xrightarrow{i} _g \theta(c'_v)\]

and

\[c_v \xrightarrow{i} c''_v \implies (\circ_M^p \circ_M^{\{x\}}) \implies \theta(c_v) \xrightarrow{\text{throw }x} _g \theta(c''_v)\]

But as we know after the exceptional transition there will be an exception record on the stack, so JVM will take the control and there will be silent transition to deal with this situation. So the case will be considered when

\[c_v = (⟨x⟩_{\text{exc}}.⟨M, p, f, s, z⟩.A; h)\]

or

\[c_v = (⟨x⟩_{\text{exc}}.⟨M, p, f, s, z⟩.c; h)\]
Case $c_v = ((M,p,f,s,z).A;h) \land i = M[p] \in \text{InvInst}$: There are two normal and exceptional possible executions. In normal execution the control will pass to the callee method and after completion it will return back to the caller context and next instruction will be fetched.

The possible receivers (one edge for each receiver) are determined by the virtual method call resolution, if the instruction is virtual invocation. If the exception reason is in the underlying callee method and there is no handler in receiver context the control passes to the caller execution context.

In exceptional case, if the reason of the exception is that the receiver object is null, then the exception will be in the caller’s context and the control transfers to the local exceptional node with $g_N =$NullPointerException tag.

Based on the rule $[\text{Inv}]$ the normal transition will be in following form:

$$i \in \text{InvInst} \land c_v = ((M,p,f,s,z).A;h) \implies ((p,i);S),c_v \xrightarrow{\cdot} ((0,i');S'),c'_v)$$

where,

$$c'_v = ((M',0,f',s',z').(M,p,f,s,z).A;h')$$

If the $g_N =$NullPointerException happens in the caller, then based on the rule $[\text{Xnv}]$ the transition will be:

$$i \in \text{InvInst} \land c_v = ((M,p,f,s,z).A;h) \land \nu((M,p,f,s,z)) = g_N \implies ((p,i);S),c_v \xrightarrow{\cdot} ((p,i);S),c''_v)$$

where,

$$c''_v = ((g_N)_{exc}.(M,p,f,s,z).A;h)$$

On the other hand,

$$\theta(c_v) = \langle c^p_M, \theta(A;h) \rangle \implies s_g = (c^p_M, \sigma)$$

Furthermore,

$$\theta(c'_v) = \langle c^0_M, \theta((M,p,f,s,z).A;h) \rangle \implies s'_g = (c^0_M, v, \sigma) \land \circ^\text{succ}_M p \in \text{next}(v)$$

(note that next determines set of target control nodes in outgoing edges)

$$\theta(c''_v) = \langle \bullet_M^{(g_N)}, \theta(A;h) \rangle \implies s''_{g_1} = \langle \bullet_M^{(g_N)}, \sigma \rangle$$

Based on graph construction rules for all $i \in \text{InvInst}$ and for all $M'$ in the set of possible receivers resulted by call resolution:

$$\langle c^p_M, \text{call } M', \circ^\text{succ}_M p \rangle \in mG(M) \land \circ^0_M \in mG(M')$$

52
3.4. CFG CORRECTNESS

and

\[(c^p_M, \text{call } M', \bullet^p_M \{ q_N \}) \in \mathcal{mG}(M)\]

Therefore for all \(i \in \text{InvInst}:\)

\[c_v^i \xrightarrow{i} c'_v \implies \left( (c^p_M \xrightarrow{\text{call } M'} \circ^p_M \text{succ } p_i) \land (c^p_{M'} \in \mathcal{E}_{M'}) \right) \implies \theta(c_v) \xrightarrow{M \text{ call } M'} \theta(c'_v)\]

and

\[c_v^i \xrightarrow{i} c''_v \implies (c^p_M \xrightarrow{\text{call } M'} \circ^p_M \{ q_N \}) \implies \theta(c_v) \xrightarrow{\text{throw } q_N} \theta(c''_v)\]

But as we know after the exceptional transitions there will be an exception record on the stack, so JVM will take the control and there will be silent transition to deal with this situations. So the case will be considered when

\[c_v = ((x)_{\text{exc}}.\langle M, p, f, s, z \rangle.A; h)\]

or

\[c_v = ((x)_{\text{exc}}.\langle M, p, f, s, z \rangle.e; h)\]

**Case** \(c_v = ((M, p, f, s, z).\langle M', p', f', s', z' \rangle.A; h) \land i = M[p] \in \text{RetInst}:\)

Based on the rule \([\text{RET}]\) the normal transition will be in following form:

\[i \in \text{RetInst} \land c_v = ((M, p, f, s, z).\langle M', p', f', s', z' \rangle.A; h) \implies ((p, i); S, c_v) \xrightarrow{\theta} ((p', i'); S', c'_v)\]

where ,

\[c'_v = ((M', \text{succ } p', f', s', z').A; h')\]

On the other hand,

\[\theta(c_v) = (c^p_M, \theta((M', p', f', s', z').A; h)) \implies s_g = (c^p_M, v, \sigma)\]

and

\[\theta(c'_v) = (c^p_{M'}, \theta(A; h)) \implies s'_g = (c^{\text{succ } p'}_{M'}, \sigma)\]

such that

\[c^{\text{succ } p'}_{M'} \in \text{next}(v)\]

(note that \text{next} determines set of target control nodes in outgoing edges)

Based on graph construction rules:

\[(c^p_{M'}, \text{call } M, c^{\text{succ } p'}_{M'}) \in \mathcal{mG}(M')\]

And also

\[c^p_M \in \mathcal{mG}(M) \land c^p_M \models r\]

53
Therefore:
\[
\forall i \in \text{RetInst} \ \exists T \in \text{res}_{\text{JBC}}^2(a,m), M = T.m \\
\implies c_v \xrightarrow{i} c'_v \implies (\circ^p_{M'}\text{call}_{M'}\circ^\text{succ}_{M'} p' \land \circ^r_{M} \xrightarrow{r} ) \implies \theta(c_v) \xrightarrow{M' \text{ret}_{M} \theta(c_v)}
\]

**Case** \(c_v = ((x)_{\text{exc}}.\langle M.p,f,s,z \rangle.\langle M',p',f',s',z' \rangle.A; h)\):  Based on the exceptional operational semantics for JVM silent transitions we have:
\[
c_v = ((x)_{\text{exc}}.\langle M.p,f,s,z \rangle.\langle M',p',f',s',z' \rangle.A; h) \implies ((p,i); S, c_v) \xrightarrow{\xi} ((p,i); S, c'_v), k \in \{1,2\}
\]
where,
\[
c'_v = ((M,t,f,s,z).\langle M',p',f',s',z' \rangle.A; h)
\]
and
\[
c'_v = ((x)_{\text{exc}}.\langle M',p',f',s',z' \rangle.A; h)
\]
On the other hand,
\[
\theta(c_v) = \langle \bullet^p_{M}\{x\}, \theta(\langle M',p',f',s',z' \rangle.A; h) \rangle \implies s_g = (\circ^p_{M}, v.\sigma)
\]
Furthermore,
\[
\theta(c'_v) = \langle \circ^p_{M}, \theta(A; h) \rangle \implies s'_{g_1} = (\circ^p_{M}, \sigma)
\]
and
\[
\theta(c'_{v_2}) = \langle \bullet^p_{M'}\{x\}, \theta(A; h) \rangle \implies s'_{g_2} = (\bullet^p_{M'}, \sigma)
\]
Based on graph construction rules for all \(i \in \{\text{XMPINST} \cup \text{INVINST} \cup \text{THRINST}\}\):
\[
\langle \bullet^p_{M}\{x\}, \text{handle}, \circ^t_{M} \rangle \in mG(M)
\]
and
\[
\langle \bullet^p_{M}\{x\}, \text{handle}, \bullet^p_{M}\{x,r\} \rangle \in mG(M)
\]
also
\[
\langle \bullet^p_{M'}\{x\}, \text{handle}, \bullet^p_{M'}\{x\} \rangle \in N^x_g \subseteq mG(M')
\]
Therefore:
\[
\left( c_v \xrightarrow{i} c'_v \implies \langle \bullet^p_{M}\{x\}, \text{handle}, \circ^t_{M} \rangle \right) \implies \theta(c_v) \xrightarrow{\text{catch}_{x}} \theta(c'_v)
\]
and
\[
c_v \xrightarrow{i} c'_{v_2} \implies \left( \langle \bullet^p_{M}\{x\}, \text{handle}, \bullet^p_{M}\{x,r\} \rangle \land \langle \bullet^p_{M'}\{x\}, \text{handle}, \bullet^p_{M'}\{x\} \rangle \right) \implies \\
\theta(c_v) \xrightarrow{M \text{ret}_{M'} \theta(c'_{v_2})}
\]

54
3.4. CFG CORRECTNESS

Case $c_v = (\langle x \rangle_{\text{exc}}. \langle M, p, f, s, z, \epsilon \rangle; h)$: Based on the exceptional operational semantics for JVM silent transitions we have:

$$c_v = (\langle x \rangle_{\text{exc}}. \langle M, p, f, s, z, \epsilon \rangle; h) \implies ((\langle p, i \rangle; S), c_v) \xrightarrow{\delta} (\langle (p, i); S \rangle, c'_{v_k}), k \in \{1, 2\}$$

where,

$$c'_{v_1} = (\langle M, t, f, s, z, \epsilon \rangle; h)$$

and

$$c'_{v_2} = (\langle x \rangle_{\text{exc}}. \epsilon; h)$$

On the other hand,

$$\theta(c_v) = (\bullet_M^{p\{x\}}, \epsilon) \implies s_g = (\bullet_M^{p\{x\}}, \epsilon)$$

Furthermore,

$$\theta(c'_{v_1}) = (\circ_M^t, \epsilon) \implies s'_{g_1} = (\circ_M^t, \epsilon)$$

and

$$\theta(c'_{v_2}) = (\bullet_M^{p\{b,r\}}, \epsilon) \implies s'_{g_2} = (\bullet_M^{p\{b,r\}}, \epsilon)$$

Based on graph construction rules for all $i \in \{\text{XMPINST} \cup \text{INVINST} \cup \text{THRINST}\}$:

$$(\bullet_M^{p\{x\}}, \text{handle}, \circ_M^t) \in mG(M)$$

and

$$(\bullet_M^{p\{x\}}, \text{handle}, \bullet_M^{b\{x,r\}}) \in mG(M)$$

Therefore:

$$c_v \xrightarrow{i} c'_{v_1} \implies (\bullet_M^{p\{x\}} \xrightarrow{\text{handle}} M \circ_M^t) \implies \theta(c_v) \xrightarrow{\text{catch } x} \theta(c'_{v_1})$$

Note here that there is no transition named $\text{abort}$ in flow graph behavior definition. We might need to extend the definition with this rule. If we extend the definition then we can have following move also:

$$c_v \xrightarrow{i} c'_{v_2} \implies (\bullet_M^{p\{x\}} \xrightarrow{\text{handle}} M \bullet_M^{b\{x,r\}}) \implies \theta(c_v) \xrightarrow{\text{abort}} \theta(c'_{v_2})$$

This concludes the proof. $\Box$
Chapter 4

Modular Flow Graph Extraction

The purpose of this chapter is to propose some basic techniques for compositional CFG extraction. In this chapter we are looking for techniques to extract CFG of modular programs compositionally. Moreover, we study how these methods can help us to extract CFG of open programs.

4.1 Introduction

In order to have a unified modular verification tool set, extracting a program’s model in a modular way is essential. Modular programs are decomposed into their parts. Modular or compositional program analysis is the area of research where the program’s analysis (semantics) is obtained from the analysis (semantics) of its modules. Scalability and dynamic evolvability can be regarded as the main purpose of these studies and they can be categorized in two groups. The first group tries to find more efficient algorithms using incremental techniques. The second group develops methods that given a program’s parts, analyze each part separately and conclude the semantics of the whole program based on the semantics of each part.

The basic solution for the first group is to analyze each module only once and incrementally. Then a so-called summary of the analysis is produced. In analyzing the other modules there is no need to re-analyze parts that have been summarized. The second group performs analysis on the modules independently. Then the semantics of the whole program is based on the compositions of the semantics of the program parts. In this method, analysis of the program parts usually needs approximation, however, in module local analysis gives an imprecise result for parts. But final step of analysis (global analysis) refines approximations. Depending on the property of analysis this composition operator needs to be defined.

Modular verification for open systems aims to employ techniques based on separate modules analysis. To build our model of a program compositionally, we need techniques to analyze each part independently and extract the complete model based on the sub-models obtained from parts. In other words compositional separate control flow graph extraction of a given Java program is about to extract the program’s
CHAPTER 4. MODULAR FLOW GRAPH EXTRACTION

CFG based on the CFG’s of parts.

Abstract interpretation formalizes the idea that the formal proof of specification satisfaction can be done at some level of abstraction where irrelevant details about the semantics and the specification are ignored [23]. Actually abstract interpretation is a partial execution of the program in an abstract universe of values. Patrick Cousot and Radhia Cousot studied compositional static analysis by abstract interpretation in [22]. The study lists a fairly complete list of previous results in different categories.

Here we shall study the composition of flow graphs which have been extracted independently. At first we review different techniques introduced in [22]. Then after considering our problem with respect to compositional analysis we will present solutions using CFG construction rules defined for closed systems.

4.2 Modular Control Flow Graph Extraction Methods

In this section we review different techniques in compositional program analysis. Then based on these techniques we propose our modular control flow graph extraction methods.

4.2.1 Modular Static Program Analysis

P. Cousot and R. Cousot in [22] grouped the modular program analysis techniques in different categories:

- Simplification-based separate analysis
- Worst-case separate analysis
- Separate analysis with user-provided interfaces
- Symbolic relational separate analysis

Simplification-based separate analysis performs a preliminary global analysis of the whole program. Then global information on the program helps the modules’ independent analysis. This approach is not considered a scalable technique for very large programs [22].

Worst-case separate analysis considers that absolutely no information is known on the interfaces. Then it separately computes the local semantics of the parts. In a final phase the global program analysis is performed employing the results of modules analysis and approximations. Obviously in this technique some parts might need modification before global analysis. This method is considered efficient but imprecise [22].

Separate analysis with user-provided interfaces assumes that the interface of each module carries some information about the properties of the module. The local analysis of modules is performed using the information provided in interfaces.
4.2. MODULAR CONTROL FLOW GRAPH EXTRACTION METHODS

Finally global analysis is done compositionally. Instead of asking the user to provide information in the interfaces, it can be provided automatically. Because some required information of modules is available through interfaces this technique can be considered as an efficient and more precise technique comparing to the worst-case analysis.

Symbolic relational separate analysis is based on the use of relational abstract domains and a relational semantics of the program parts [22]. In this approach all external objects used or modified in part $P_i$ are given a symbolic name. Any action on these external objects is performed in a lazy way and the possible effects are postponed to when the actual objects are known. In composition of parts the external names are bound to the actual objects. This approach is claimed to be very effective when dependency graph of modules is known [22]. Because in this technique each part is executed using symbolic objects instead of real values, this technique is also known as symbolic execution.

4.2.2 Modular Java Programs

A given Java application is partitioned into sub-systems defined as packages containing a set of Java classes. Each class can be studied as an independent unit encapsulating the behavior as the implemented code for methods and encapsulating the state as the attributes of the class. Classes can be the minimal units of a Java program that might exist. Let us suppose our granularity of program separation is Java packages. In the worst-case each unit will be a single class, but usually applications or libraries are subdivided into separate sub-systems containing a set of classes and can be separately developed. Also design of the applications intend to have loosely coupled modules or sub-systems.

So in general a given Java program $P$ is made up from a set of sub-programs or packages $[P_1, P_2, ..., P_n]$. The compositional separate modular static analysis of the program $P$ will be based on the separate static analysis of its parts $P_i$. Each part $P_i$ is either an open or closed system.

Based on the program model, components are supposed to communicate through method calls. If a class from $P_i$ has a has-a (association) or is-a (generalization) relationship with any class of $P_j$ then $P_i$ and $P_j$ have structural dependencies. We say two modules $P_i$ and $P_j$ have behavioral dependencies if the control flow of execution from a method of any class of $P_i$ or $P_j$ passes to a method of any class of $P_j$ or $P_i$. Behavioral and structural dependencies between modules cause control flow analysis of module $P_i$ to be dependent upon the knowledge of the other modules $P_j$, $i \neq j$. In following we will use function $cs : \text{PACK} \rightarrow \mathcal{P}(\text{CLASS})$ to determine the set of classes declared in a given package.

4.2.3 Simplification-based Compositional CFG Extraction

Simplification approach assumes the existence of the complete program. In order to have compositional analysis following the simplification-based approach it is nec-
necessary to have a closed system. In a closed system all parts of the program are available through their implementations. To extract the CFG of each module, we need knowledge about dependencies and hierarchical information from modules. A pre-analyze process can help us to collect these information.

In separate CFG extraction only method calls (possible receivers in virtual method calls and exceptions propagations) are supposed to take advantages of the global information. If CHA is assumed as the algorithm for virtual method call resolution (see section 1.2.1) then hierarchical dependencies among the classes and methods membership information suffice to have a precise approximation in virtual call edges. So sub-typing information and methods signatures declared in every class will be the result of pre-analysis process in case of CHA. Exceptions can be propagated using iterative algorithms to find the least fixed-point.

But if the algorithm RTA is going to resolve the virtual method calls (see section 1.2.1), in addition to the information required for CHA, pre-analysis process needs to search for all instantiated classes inside methods body. This makes simplification-based technique inefficient in case of RTA, because the methods body needs to be parsed twice: looking for instantiated classes and CFG construction.

This approach is not applicable for open systems in which some of the modules are available through their interfaces. In case of closed systems the rules developed in 3.3.3 can be employed to construct the CFG of the program. If the hierarchy and class membership are considered as the preliminary global analysis of the program, our developed rules use these global analysis results to build the CFG. We implemented this technique as our CFG extractor tool. We will discuss more about the implementation in chapter 5.

### 4.2.4 Worst-case Compositional CFG Extraction

Now the problem is how to construct control flow graphs for the methods of classes in a package following the worst-case approach. Worst-case analysis does not perform preliminary global analysis because it assumes there might be an incomplete program. So analyzing a module there are two possible cases to happen. If required knowledge for local analysis exists, so we analyze it. But if there is no knowledge while analyzing, then we make an approximation. Then based on this approximation we build our control flow graph. Approximations in local analysis might be imprecise for external objects but global analysis is responsible for refinement.

Composition of the components will shed light on the hidden dependencies. Therefore global analysis of the program is able to perform modification of elements with approximations based on the new knowledge discovered from components. Thus in global control flow graph construction we build up the program’s CFG by composition of packages CFGs and composition process has to refine our approximations.

Now assume that a class \( C_i \in cs(P_i) \) is inherited from a class defined in a different package like \( C_j \in cs(P_j) \). Also the child class has overridden all parent’s methods. So we know that \( \Gamma \vdash C_i <: C_j \) or \( C_i \in subT(C_j) \) (\( \Gamma \) is the environment for
4.2. MODULAR CONTROL FLOW GRAPH EXTRACTION METHODS

Extracting CFG of methods inside $P_j$ will find $C_j$ as the only possible receiver in virtual call edges (invokevirtual($o,M$)) based on the partial information if $C_j$ be static type of the object ($C_j = staticT(o)$). So the composition of two modules should heal the edges being affected by new information. Adding $C_i.m$ as another possible receiver of virtual call and exception handling edges for exceptional returns from $C_i.m$ are parts of approximation refinement.

In our problem statement modifications are only reasonable on edges labeled as method calls. Collecting possible exceptional returns from the callee is one of the modifications that refinement process needs to perform. If there is no edge labeled with calls or the methods graph is well-formed (there is no exceptional return node) then global analysis does not require module semantics modification.

The type of the receiver object in virtual method calls will be another modification caused by new hierarchical dependencies discoveries. In modules’ composition if there is no method call or new hierarchical relationship then there will be no need for refinement.

Based on our definition (see Definition 10) of a program $P$, $\mathcal{V}$ is set of nodes which are constructed implicitly by constructing set of edges. i.e. $\mathcal{E}$. In general form of a solution for compositional CFG extraction we need to define a composition operator, denoted as $\oplus$, on locally extracted CFGs. For each module, there is a subset of edges which contains properties that have to be approximated in a local analysis, meaning that their analysis depends on information from other modules. Let us name this subset of edges as $\mathcal{E}^- \subseteq \mathcal{E}$ in which we select (-) to show that elements of the set are not complete and the set requires refinement in global analysis. Therefore subset of edges that do not need approximation in local analysis (are not dependent on external objects) are obtained simply by removing $\mathcal{E}^-$ from the whole set, i.e. $\mathcal{E}^+ = \mathcal{E} \setminus \mathcal{E}^-$. Intuitively, in order to obtain the composition of two module’s graphs we separate elements of each module into elements not involved in re-analyzing (elements which can be completely analyzed based on the local information) and elements possibly affected by new environment. The composition of two modules must refine dependent elements. Union of independent elements and the result of the dependent element refinement builds global analysis result. Now based on the fact that composition operator, i.e. $\oplus$, is associative, we can define the result of the CFG composition for two sub-program or modules (packages) as follows:

$$(\Gamma_1 \cup \Gamma_2) \vdash G(P_1) \oplus G(P_2) = (\mathcal{E}_1^+ \cup \mathcal{E}_2^+) \cup (\mathcal{E}_1^- \oplus \mathcal{E}_2^-)$$

where

$$(\Gamma_1 \cup \Gamma_2) \vdash \mathcal{E}_1^- \oplus \mathcal{E}_2^- = \gamma^\alpha(\mathcal{E}_1^-) \cup \gamma^\alpha(\mathcal{E}_2^-)$$

Edges labeled with method calls are the only elements possibly be affected by modules composition and might need refactoring. Virtual method call resolution and exception propagation are our refactoring actions in CFG composition.
Edges that do not indicate method calls (modules composition does not affect them) are considered as independent elements in CFG composition. Modification refines our approximations on edges carrying method calls, which is accomplished by $\gamma^\alpha$, within new environment $(\Gamma_1 \cup \Gamma_2)$ and employing algorithm $\alpha$ for virtual method call resolution. In the sequel we define our refinement function $\gamma^\alpha$ for two known algorithms $\alpha = \text{CHA}$ and $\alpha = \text{RTA}$.

**Refinement in case of $\alpha = \text{CHA}$**

The following describes the result of re-analyzing by $\gamma^\alpha$ in new environment for $\alpha = \text{CHA}$ as resolution algorithm. Our approximation in local analysis for virtual calls, if $\alpha = \text{CHA}$, can be the static type of the object being called. If we construct CFGs’ of methods declared in classes of module $P_i$, $T = \text{staticT}(o)$ is our first answer for any $\text{invokevirtual}(o,M)$ in method’s body of any class inside $P_i$. Moreover if there is any class $T' \in \text{cs}(P_i)$ such that $\Gamma_i \vdash T' <: T$ then $T'$ is also considered as another element of our approximated set of possible receivers. So the initial values for set $\mathcal{E}^-_i$ will be:

$$\mathcal{E}^-_i = \{ (\circ^p_M, \text{call } T.N, o^\text{succ}_M^p) \mid M \in \text{meths}(C_k), C_k \in \text{cs}(P_i) \}$$

where $M[p] = \text{invokespecial } (T,N)$ or $M[p] = \text{invokestatic } (T,N)$ or

$$M[p] = \text{invokevirtual } (o,N) \land \Gamma_i \vdash T <: \text{staticT}(o) \land \text{lookup}(n,T) = \text{sig}(n)$$

Recall that based on our convention $N$ indicates signature of method $n$. Assume that composition of all modules builds complete program environment $\Gamma = \bigcup_{1 \leq k \leq d} \Gamma_k$ (program is divided into $d$ packages). Application of $\gamma^\text{CHA}$ on $\mathcal{E}^-_i$ in new environment $(\Gamma)$ preserves initial values for $\mathcal{E}^-_i$ and tries to add more edges for possible receivers, say $\mathcal{R}_i$, found in new environment. We need also to consider that adding any possible edge for virtual call might add set of edges, say $\mathcal{X}_i$ as the result of exceptional returns from new possible callees. So we define refinement for $\mathcal{E}^-_i$ in new environment as follows:

$$\Gamma \vdash \gamma^\text{CHA}(\mathcal{E}^-_i) = (\mathcal{E}^-_i \cup \mathcal{R}_i \cup \mathcal{X}_i)$$

where $\mathcal{R}_i = \{ (\circ^p_M, \text{call } T'.N, o^\text{succ}_M^p) \mid (\circ^p_M, \text{call } T.N, o^\text{succ}_M^p) \in \mathcal{E}^-_i \}$

such that:

$$\Gamma \vdash T' <: T \land \text{lookup}(n,T') = \text{sig}(n)$$

$\mathcal{X}_i$ indicates propagated exceptional edges in new environment. Edges for handling $g_N$, which symbolizes possible NullPointerException for object being called, are constructed in local CFG construction and new environment does not affect
4.2. MODULAR CONTROL FLOW GRAPH EXTRACTION METHODS

them. But we have to handle all uncaught exceptions in newly found receiver and add corresponding edges as our refinement process. So in order to define $X_i$, we need the set of the signatures of all methods that are invoked in $R_i$ and $E_i^-$. We define this set as follows:

$$\mathcal{N} = \{ N \mid (\circ_M^P, \text{call } T.N, \circ_M^{\text{suc}} P) \in (R_i \cup E_i^-)\}$$

Then $X_i$ will be set of all exception handling edges for all exceptions propagated through calls in $\mathcal{N}$ and is defined as follows:

$$X_i = \bigcup \left\{ \left\{ (\circ_M^P, \text{handle}, \circ_M^x) \right\} \cup H'^x \mid \bullet_{N_k}^{(x,r)} \in m\mathcal{G}(N_k), N_k \in \mathcal{N} \right\}$$

where

$$H'^x = \left\{ \begin{array}{l}
\Gamma \vdash x : y \implies \\
\left\{ \begin{array}{l}
\{ (\circ_M^p, \text{handle}, \circ_M^x) \} & h_{\Gamma[M]}(p, y) = t \neq 0 \\
\{ (\bullet_M^p, \text{handle}, \bullet_M^x) \} & h_{\Gamma[M]}(p, y) = 0
\end{array} \right. 
\end{array} \right. \right.$$

In summary, if a new sub-type of any possible receiver is found in new module then it will be added to the possible receivers together with all uncaught exceptions in new callee. Exception propagation is possible using iterative work-list algorithms to find the fixed-point because domain of exceptions is finite and exception propagation is monotonic. Again to be efficient, call graph$^1$ can be constructed in compositions to perform algorithm in reverse topological order.

In order to have an efficient CFG extractor refinement process tends to not to re-analyze new module to find the required information. We desire to analyze the code of each module only once. So we need to keep information that might be useful in global CFG extraction. As we can see from the rules, the key information for normal CFG extraction using algorithm CHA is hierarchical and method membership information. The result of local CFG extraction is realized in CFGs of each class, so method membership information can be obtained if we have CFGs of a class.

Global CFG extraction needs hierarchical dependencies of classes and exception handling table to perform refinement. Furthermore it is desirable to exploit the results of module’s analysis. Therefore here we need to extend the definition of basic method specification and propose definition of class specification.

Intuitively we define class specification to carry class structural information and methods exception handling table. First, we equip method specification with exception handling table. Global analysis uses the method’s exception handling table for exception propagation. We recall here that every method in JBC is denoted as $\Gamma[M] = (P, H)$ such that $P$ shows the body code and $H$ is a partial map of addresses to handlers. Now we are going to extend methods specification to carry exception handling table.

$^1$Call graph of a program is a directed graph that represents the calling relationships of the program methods.
Definition 15 (Method Exception Specification for CHA) Specification of a method \( M \) is a triple \((M_M, E_M, X_M)\) where

- \( M_M = \text{mg}(M) \)
- \( E_M \) is a non-empty set of entry points for \( M \).
- \( X_M = H \)

In next step we define specification of a class. Intuitively specification of a class consists of its parent class and the set of all specifications of methods declared within the class indexed with method signature.

Definition 16 (Class Specification for CHA) Specification of a class \( C \) is a pair \((T, S)\) where

- \( T = \text{subT}(C) \) and
- \( S = \{(M,(M_M, E_M, X_M)) \mid M \in \text{meths}(C)\} \)

Now based on definition 16 we can save the result of a class analysis in the class specification. Actually class specification determines semantics of the class in CFG extraction. Now if we re-define function \( h \) for \( X_M \), in composition of every two classes, as the minimum unit of composition, we will have all required knowledge to extract the compositional CFG. It is obvious that in final step if the composition of all modules does not build a closed system, elements dependent on unavailable parts (external objects) will remain with their approximated values, resulting to possibly imprecise consequences.

We should note here that if a class is not available through its implementation and it has not been implemented, \( X_M \) for every method of the class will be empty. It happens because exception handling information will not be available if the method instructions have not been specified. We will discuss this case in user-provided CFG extraction technique.

Refinement in case of \( \alpha = RTA \)

Recall here that RTA algorithm tries to prune classes which are not instantiated in the program from the set of reachable types resolved by CHA. Pruning process is not a kind of action that we can do incrementally in each pair composition. If a type is pruned in the composition of two modules, and another module presents it as instantiated type, then there is no way to restore it. This process must be executed in global analysis, when we have all sets of instantiated types in modules.

We can see that RTA-based global analysis, in addition to hierarchical information, needs to know about instantiated types of each module. So at first step we have to extend definition of class specification and equip it with types instantiated
inside methods and fields\(^2\). We do not need to save instantiated types of a method because it is deducible using method’s CFG. Every edge labeled with a class constructor call can be considered as a type instantiation. Intuitively, specification of a class consists of its parent class, union of all specifications of methods declared in class and set of all classes instantiated in its fields.

**Definition 17 (Class Specification for RTA)** Specification of a class \(C\) is a triple \((T,S,I)\) where

- \(T = \text{subT}(C)\) and
- \(S = \{(M,(\mathcal{M}_M,\mathcal{E}_M,\mathcal{X}_M)) \mid M \in \text{meths}(C)\}\)
- \(I = \{T \mid T \text{ is instantiated in field of } C\}\)

CFG extraction using RTA algorithm separate elements indicating virtual method calls from static and constructor calls. Because we cannot keep values of approximations for virtual calls, refinement process needs to decide about these elements separately. We define static type of object being called as our approximation for virtual calls. Types of objects for constructor and static calls are known at the time of local analysis. So initial value for \(E^-_{ic}\) will be edges carrying constructor and static calls \(E^-_{ic}\) plus set of virtual method calls \(E^-_{iv}\). Following formalizes elements of \(E^-_{ic}\):

\[
E^-_{ic} = E^-_{ic} \cup E^-_{iv}
\]

where

\[
E^-_{ic} = \{(\circ^p_M, \text{call } T.N, \circ^\text{succ}_M) \mid M \in \text{meths}(C_k), C_k \in \text{cs}(P_i)\}
\]

such that

- \(M[p] = \text{invokespecial } (T,N)\) or \(M[p] = \text{invokestatic } (T,N)\)

and

\[
E^-_{iv} = \{(\circ^q_M, \text{call } T.N, \circ^\text{succ}_M) \mid M \in \text{meths}(C_k), C_k \in \text{cs}(P_i)\}
\]

such that

- \(M[q] = \text{invokevirtual } (o,N) \land T = \text{staticT}(o)\)

Now we are ready to define refinement of edges labeled with method calls using RTA. First of all we refine each subset of method calls separately in global analysis and define \(\gamma^{RTA}(E^-_{ic})\) in new environment \(\Gamma\) as follows:

\[
\Gamma \vdash \gamma^{RTA}(E^-_{ic}) = \gamma^{RTA}(E^-_{ic}) \cup \gamma^{RTA}(E^-_{iv})
\]

\(^2\)In Java it is possible to create an object as field member and use the created object inside methods.
\( \gamma^{RTA} \) preserves elements of \( E^-_{ic} \) and tries to propagate exceptions through method calls. So we will have:

\[
\Gamma \vdash \gamma^{RTA}(E^-_{ic}) = (E^-_{ic} \cup \mathcal{E}_i)
\]

where \( \mathcal{E}_i \) shows all exceptional paths resulted as exceptions propagation through non-virtual calls. To build this set we collect the signatures of all methods been invoked in non-virtual call edges \( (E^-_{ic}) \). We name this set as \( \mathcal{N}_c \) and define it formally as follows:

\[
\mathcal{N}_c = \{ N \mid (\circ p_M, \text{call } T.N, \circ_p \text{succ } M) \in E^-_{ic} \}
\]

Then for all methods in \( \mathcal{N}_c \) we define exceptions propagation as follows:

\[
\mathcal{E}_i = \bigcup \left\{ \{ (\circ p_M, \text{handle}, \bullet p_M) \} \cup \mathcal{H}^{tx} \mid \bullet q_{N_k}^{(x,r)} \in mG(N_k), N_k \in \mathcal{N}_c \right\}
\]

such that

\[
\mathcal{H}^{tx} = \left\{ \begin{array}{ll}
\{ (\bullet p_M^{(x)}), \text{handle}, \circ_p M \} & \text{if } h_{\Gamma[M]}(p,y) = t \\
\{ (\circ p_M^{(x)}), \text{handle}, \bullet p_M^{(x,r)} \} & \text{if } h_{\Gamma[M]}(p,y) = 0
\end{array} \right.
\]

Virtual call resolution using RTA requires knowledge about instantiated objects. So at first we have to define the set of types instantiated inside methods body. Since we desire not to analyze instruction set of method again, we collect instantiated types using edges labeled with object constructor calls. In JBC constructors are named \textit{init}. So we need to extract name of a method call label in edges and if the name is equal to \textit{init}, it is implied that a constructor is called and the type is collected as instantiated class. Following define the set formally. Recall that \( N \) is signature of a method with name \( n \).

\[
IC = \{ T \mid (\circ p_M, \text{call } T.N, \circ_p \text{succ } M) \in mG(M) \land n = \text{init} \}
\]

Similarly we define set of types instantiated in fields of classes using class specification. Fields of a class is constructed only of its constructor is called. So we have:

\[
IC_f = \{ T \in IC \mid C \in IC \}
\]

The complete set of all instantiated types is defined as follows:

\[
IC^\Gamma = IC_f \cup IC
\]

\( \gamma^{RTA} \) is not able to preserve elements of \( E^-_{iv} \). Here refinement process resolves virtual calls and then propagates all exceptions through resolved call edges. So we define refinement as follows:

\[
\Gamma \vdash \gamma^{RTA}(E^-_{iv}) = (\mathcal{E}_i \cup \mathcal{E}'_i)
\]
4.2. MODULAR CONTROL FLOW GRAPH EXTRACTION METHODS

where $\mathcal{R}_i$ shows the set of all RTA resolved possible receivers in virtual method calls ($\mathcal{E}_{i_v}^-$) and is defined formally as follows:

$$\mathcal{R}_i = \{(p_M^c, \text{call } T', N, v^{\text{succ } p}) | (p_M^c, \text{call } T.N, v^{\text{succ } p}) \in \mathcal{E}_{i_v}^-\}$$

such that:

$$\Gamma \vdash T' : T \land \text{lookup}(n, T') = \text{sig}(n) \land T' \in IC_{\Gamma}$$

and also $\mathcal{X}'_i$ shows all exceptional edges resulted as exception propagation for each call edge appears in $\mathcal{R}_i$. To define this set formally we collect the signatures of all call edges in $\mathcal{R}_i$ as below:

$$\mathcal{N}_o = \{N | (p_M^c, \text{call } T.N, v^{\text{succ } p}) \in \mathcal{R}_i\}$$

Then we need to collect all exceptional returns from these methods and build our exceptional paths. We define the propagated exceptional paths formally as follows:

$$\mathcal{X}'_i = \bigcup \{(p_M^c, \text{handle}, \bullet_{M}^{{\{x\}}}) \cup \mathcal{H}^{\text{tx}} | \bullet_{N_k}^{{q_{r,\{x,r\}}}} \in \mathcal{mG}(N_k), N_k \in \mathcal{N}_o\}$$

where

$$\Gamma \vdash x : y \implies \mathcal{H}^{\text{tx}} = \{\{(p_M^c, \text{handle}, \bullet_{M}^{{\{x\}}})\} \cup \bullet_{N_k}^{{q_{r,\{x,r\}}}} \in \mathcal{mG}(N_k), N_k \in \mathcal{N}_o\}$$

4.2.5 Compositional CFG Extraction: User-Provided Interfaces

In compositional analysis based on user-provided interfaces specification of abstract units needs to be defined in a proper formal language precisely. CFG extraction process can refer to abstract components specification for external objects. In basic CFG extraction (CFG without exceptional elements), possible receivers in method calls can be resolved by analyzing hierarchical information for CHA and class instantiations for RTA. Specification of missing components needs to provide information required for virtual method call resolution algorithms.

Currently compositional verification tool set for open systems uses a fragment of propositional $\mu$-calculus [26] and temporal logics like LTL [27] to specify the control flow safety properties of components [13]. Every method is annotated with this specification. The specification of a method together with its control flow graph build the method’s interface. In order to use user-provided interfaces technique for CFG extraction we need to have richer interfaces. To achieve these rich interfaces we have to extend them to carry hierarchical information defined in definition 16 for CHA or definition 17 for RTA. Then our CFG extraction based on worst-case approach is applicable for these abstract components.

Exceptional properties of an implemented method are due to instructions set used in implementation. So if the abstract component is not finalized with respect
to implementation, expressing exceptional properties will not be possible. Therefore we are not able to extract exceptional flows of CFGs based on user-provided interface techniques.

Here we propose an idea in user-provided interface CFG extraction based on models available for abstract components. Normally if a component is not available yet through its implementations, we can take advantages of design models, like class diagrams and sequence diagrams in UML. Class diagrams contain hierarchical and class members information profitable for CHA based approximations. Sequence diagrams also can be analyzed to extract all possible class instantiations inside methods. So sequence diagrams along with class diagrams make it possible to provide required information about external objects in CFG extraction. Moreover, sequence diagrams of UML 2.0 can be analyzed to obtain control flow information of abstract components [28].

### 4.2.6 Compositional CFG extraction: Symbolic Execution

Compositional CFG extraction can take advantage of using symbolic execution approach to determine possible receivers in method calls. In modules local analysis assigning static types of objects as their values and executing the methods can determine possible values of object in method calls. In symbolic execution the values of objects in some points depends on the return values of a method or type of the parameter passing to the method. In these cases the external actions can be postponed to global analysis when the missing components are known. To accomplish the effects of external methods on object types, methods summary graph [24] can be used.

Whaley and Lam use method summary graph for points-to analysis [24]. The method summary graph carries information about input parameters of every method, their effects on local variables as read and write actions and finally it specifies method’s return type. Summary transfer function [25] is also useful notion to find methods effects on objects values in symbolic executions.

In general symbolic execution of methods needs to express its effects on external objects by building a summary method graph. Global analysis can use these methods summary graphs to conclude about unresolved types of method call edges. Uncaught exceptions of each method call needs to be delayed in global analysis. When all call edges resolved by their possible receivers, uncaught exceptions can be propagated to callers in an iterative procedure.

### 4.3 Compositional CFG extraction: Hybrid Approach

Here we propose a hybrid approach for compositional verification of open systems. In this approach a combination of techniques that have been explained so far assists us to benefit of every technique. Symbolic execution of methods gives a more precise approximation for possible receivers in method call edges. For those components available through their implementations we use worst-case analysis to analyze each
component independently. In order to have a more precise approximation, since every method’s instruction set needs to be parsed for CFG extraction rules, symbolic execution of method can be parallelized with CFG extraction process. We can build CFG and method summary graph in parallel and define semantical relations between unresolved types of methods calls and symbolized objects in method summary graph.

Composition of local semantics and global CFG extraction can take advantages of user-defined interfaces for abstract components, methods summary graphs and class specifications obtained in local analysis. In every external object type resolution, if the component is available through its implementation (concrete component) then we can use its analysis summary and classes specification information. If the component is available through its interface (abstract component), then the global analysis compulsorily needs to extract information via the user-provided interfaces.
Chapter 5

Flow Graph in JBC: Implementation

The CFG extraction rules defined in the previous chapters are implemented in the OCaml programming language. This chapter introduces libraries employed to handle byte-code information. Then two versions of programs (complete and incomplete systems) for CFG extraction are explained. In this chapter we presume that the reader has basic knowledge about the OCaml programming language.

5.1 Introduction

The tool set developed for compositional verification [3] is available in the OCaml programming language. To have a unified tool set for program model extraction and verification we select the OCaml language as our implementation programming language. On the other hand, to implement the control flow graph extractor from the Java byte-codes an efficient library to manipulate .class files is an essential component. Javalib [30] parses Java byte-code classes (files with .class extensions) and provides information in OCaml [29] data structures. This library makes it possible to extract byte-code information, manipulate and produce valid byte-code information in .class format.

Sawja [30] is a library developed on top of the Javalib to build some ready to use and fundamental code static analysis components. Virtual method call resolution and transformation to a stack-less intermediate language called BIR [32] are major features of the Sawja library. [31] lists main features of Sawja as below:

- parsing of .class files into OCaml data structures and unparsing of those structures back into valid .class files

- decompilation of the bytecode into a high-level stack-less intermediate language called BIR

- sharing of complex objects both for memory saving and efficiency purpose

- the determination of the set of possible receivers in virtual method call resolution (using several algorithms, including RTA)
CHAPTER 5. FLOW GRAPH IN JBC: IMPLEMENTATION

- a careful translation of many common definitions of the JVM specification
- manipulating abstract data structures representing native methods object allocation and method calls

To have knowledge about current available libraries suitable for Java byte-code analysis reading of [31] is suggested. Based on our current knowledge Sawja is the only efficient library in OCaml available and suitable for our requirements. Thanks to a complete and comprehensive documentation of the libraries, here we will not explain how to use or install these libraries. A tutorial to start coding of an analyzer and also all complete information about APIs are available at [30]. Figure 5.1 depicts a simple schema of an analyzer placement using Javalib and Sawja.

![Figure 5.1. Javalib and Sawja](image)

### 5.2 CFG Extractor

CFG extractor program is developed in two separate versions. Apart from the efficiency problem of the current version of the program analyzer, which is developed based on Soot [4], it is not able to extract the CFG in modular way. The purpose of the first version is to replace the current program analyzer being used in compositional verification tool set. In this version the simplification-based method of modular CFG extraction (4.2.3) is implemented and a complete JBC program is assumed. So user needs to provide the path for Java classes, third party libraries and program. Also introducing main entry class and entry method is basic requirement. In this version CFG extractor asks Sawja to parse and build (pre-analysis)

---

1. This is latest reported feature updated on Fall 2010.
the complete program including all libraries. Sawja builds the complete program and call graph, thus provides functions to find possible receivers given a call point. Then our developed CFG extractor takes the advantages of the JProgram built by Sawja and construct CFG for methods from our program which are reachable from specified entry method. If there is not a possible path of execution for a method \( M \) from the program being analyzed, then \( mG(M) \) will not be constructed. Figure 5.2 depicts different layers of this extractor.

![Figure 5.2. Complete Programs CFG Extractor](image)

The second version of CFG extractor is developed to parse a sub-program with the purpose of implementation of the other compositional CFG extraction techniques, i.e. worst-case, user-provided interfaces and symbolic execution. The final goal for this version is to extract CFGs of a set of incomplete subprograms and obtain global CFG by compositions of part’s CFGs. Because we don’t have a complete program, Sawja is not used in this version and analyzer tries to employ Javalib to build sub-programs from a set of .class files. If the parser encounters a missing element during constructing of relationships, ignores it with the hope of finding the missing part in composition phase. CFG constructor also makes approximation for external objects (objects are not available in module being analyzed) and final phase refines all approximated elements. Figure 5.3 shows layers of this version.

![Figure 5.3. Modular CFG Extractor](image)

Our complete programs CFG extractor can be considered as closed system ana-
lyzer because it analyzes the whole program and does not permit to have a missing element. But from CFG construction point of view we cannot name it as closed system graph extractor. When a graph extractor finds a call edge targeting a class of Java library, it does not construct CFG of the class outside the program scope. In fact CFG constructor for complete program draws a border between the program and libraries used by the program. On the other hand second version of the program from the CFG extraction point of the view (analysis and CFG construction) is developed for a set of open (sub-)programs. So we cannot distinguish these two versions of programs using concepts of closed/open systems. Let name the first version as ComFGExtractor pointing for complete programs and the second version as ModFGExtractor indicating modular flow graph extraction. In following these two programs are explained separately.

### 5.3 Complete Program CFG Extractor

So far we have explained that global CFG extractor needs to have a complete set of elements and we named it as ComFGExtractor. Figure 5.3 shows general picture of procedure performed by ComFGExtractor. The process starts by loading required settings. Then JRTA \(^2\) is asked from Sawja to build the program specified by user. The set of reachable methods (parsed methods) will be our working set. CFG construction rules are applied for each reachable method defined inside a class which is also a member of the program. Following explains essential functions of the program.

#### Starting Point

Like any other programs we start by main function. As usual function main tries to set the required parameters and start the whole process. The first argument is supposed to be settings file. Function load_settings given settings file loads parameters needed by CFG extractor. The result will be in variables Jlib, for JDK library path \(^3\), olib, for third party library path, plib, for program path, ec, for entry class and finally em, for entry method of the whole program. Having set output format in out_format (KTH and Normal are two possible formats), control flow graph extraction process is started by rta_cfg_extract and the final result will be printed out either in standard output or in a file specified by the user.

---

\(^2\)Any other module like JCRA, JRRTA, XTA could be used. We selected RTA as a proper and safe approximation for our virtual calls resolution.

\(^3\)This might be confused with Javalib. Javalib refers to a library developed in OCaml to analyze JBC programs. JLib in settings file refers to the path of Java classes installed in the system.
5.3. COMPLETE PROGRAM CFG EXTRACTOR

```ocaml
let rta_const_cfg prg entry graph id =
let rec const_cfg pp cfg =
  let cfg_node p = ... (*build unique node for p*) in
  let op_code = ... (*op_code in pp*)
  and norm_succ = ... (*normal successors*)
  and exp Succ = ... (*exceptional successors*)
  ...
  in
match op_code with
| OpInvoke ((‘Virtual o),cms) -> begin ... end
...
| OpReturn -> begin ... end
in
const_cfg entry graph

let rta_prgr_cfg prg refined_classes refined_meths=
let get_class_cfg cn id =
  let get_meth_cfg ms cid mid =
    (rta_const_cfg prg first pp CFGSet node_prefix)
  ... in ...
(*get_meth_cfg for all methods in cn*)
in ...
(*get_class_cfg for all classes in prg*)

let rta_cfg_extract prg_path lib_path entry_class entry_meth format =
let dir = ... (*concatenation of all paths*)
and cnss = read_classes prg_path in
let class_map = (load_classes prg_path cnss) in
let prg_meths = (get_prg_cmsig class_map) in
let entry_sigs = ... (*entry method signature*)
in
let (prta,_) = JRTA.parse_program dir entry_sigs in
let ref_meths = ... (*intersection of prd_meths and prta.parsed_methods*)
and ref_classes = ... (*intersection of load_classes and prta.classes*)
in
(rta_prgr_cfg prta ref_classes ref_meths)

let main =
let (jlib,olib,pplib,ec,em)= load_settings set_file in
let result = rta_cfg_extract prg_path lib_path ec em out format in
...
```

Figure 5.4. OCaml Summary Code: CFG Extractor for Complete Programs

CFG Extraction using JRTA

Function `rta_cfg_extract` starts by extracting all class names from the files appeared in the program path and invokes `load_classes` to load class information in a class map. `ClassMap` is a module implemented in `Javalib` and makes it possible to build a map of elements (for example class information) indexed by class names (values of type `class_name`). Then given map of classes in the program directory indexed by class names, function `get_prg_cmsig` makes a list of methods signatures defined inside the loaded classes. The list is only the names of concrete methods (methods that contain implementations) and abstract methods are not considered by this function. As we mentioned previously the simple name of the entry method is passed to the extractor (`.prta_cfg_extract`). `get_signature` tries to fetch the entry method signature and if specified entry method is not found assumes that method `main` should be defined in the entry class and keeps it as default entry point.
of the program. Now entry point of the program and all required directories are provided and we can ask module JRTA to load the program employing RTA algorithm for virtual method call resolution.

Thanks to the reachability analysis performed by JRTA.parse_program there might possibly be still some unparsed methods. Also in order to prune methods and classes outside program's scope, intersection between methods and also classes parsed by the JRTA and the ones loaded inside the extractor is obtained (ref_meths and ref_classes). Now everything is ready to extract control flow graphs by rta_prg_cfg and convert them to string by prg_cfg_to_str.

Loading Classes

Function load_classes simply given a list of class names uses Javalib.get_class to read the class information and adds it as an element in a class map with corresponding class_name. Javalib.get_class given the path and the class name parse related .class file and builds information for jclass or jinterface. Our assumption is that we only have classes and this function does not check if it is an interface or class. There could be a type checking on variable c to build two separate maps of classes and interfaces.

Program CFG

Function rta_prg_cfg tries to decompose the program into classes and extract the graphs by extracting flow graph for each parsed methods inside each class. prg_cfg given a list of class names extract class control flow graph and makes a map of set of CFGs indexed by class name. Each class is coded by an id generated by cid_succ. Control flow graph for a class is a disjoint union of control flow graphs of methods defined in the class. So get_class_cfg given a class name asks JProgram.get_node to find the node in parsed program. Then fetches the list of all concrete methods inside the class calling JProgram.get_concrete_methods. Passing the list of concrete methods and corresponding class id , recursively extract control flow graphs of methods (class_cfg). This is achieved by calling get_meth_cfg. Like class id every method is coded with an id generated by mid_succ in each step. get_meth_cfg uses module JControlFlow.PP to find the first control point of the method and invokes rta_const_cfg to construct the control flow graph. Pair of class id and method id is given to this construction method to build the nodes of the graph.

CFG Construction Rules

Function rta_const_cfg is the construction engine of the control flow graph extractor based on the rules defined. It explores all the instructions in the method body and establishes the edges between each two sequence of control points. This function gets the first program pointer of the method which should be the instruction labeled with address 0, then asks a recursive constructor name const_cfg to build the complete CFG of the method.
const_cfg for each given program pointer finds the instruction, normal and exceptional successors of the current control point using JControlFlow. Then based on the instruction type (op_code) establishes the edges between two nodes. Each node is structured uniquely by cfg_node.

General procedure to each instruction is to find the next program pointer, makes labels based on the current instruction, builds the edge employing cfg_edge, add the edges between current node and successors to the set of edges for method. This process is repeated for next pointer until set of instructions in method body finishes. Termination decision is made when the operation code is of type OpReturn and there is no more instruction to continue. In OpReturn case if there is any instruction left, construction process continues otherwise an exception is thrown by JControlFlow.next_instruction and as a result of handling thrown exception, constructed graph is returned.

Construction rule for most of the instructions looks similar. Except operation code of virtual method call in which multiple edges between two nodes must be established. static_lookup_method implemented in module JProgram resolves virtual method call and gives all possible receivers discovered by algorithm RTA (program is loaded with JRTA). If the set of possible receivers resolved by RTA is empty, program tries to find possible receivers using CHA algorithm. This adaptive approach in virtual call resolution gives the CFG extractor power of approximating call edges for special types of open systems in which concrete components do not have outgoing call to abstract components, like applets in mobile codes. In this type of open system abstract component is supposed to provide inputs to concrete one and ask for service. RTA algorithm fails to analyze this type of open system, because it is not able to find any instantiations of objects being sent for applets (instantiations are in missing module). We use adaptive resolution approach and in case of RTA failings, tool switches to CHA approximation.

Function add_cfg_edge given source node (s), list of target nodes (t1), transition edge (e) builds every single edge and adds to the given set of edges. Control flow graph for a method is considered as a set of edges of type t_cfg_trans.

Output Format

Function prg_cfg_to_str represents set of graphs for a given program as string based on the given format KTH or Normal. For KTH, cfg_to_kthstr is used and for Normal, cfg_to_string is employed.

Normal format is just to print out the graph in a simple readable format. In this format the graph is printed as a set of edges like (n,e,n') in which n is the source node, n' is the target node and e is the label of the edge which should indicate corresponding instruction.
Program Usage

The program needs five parameters to start. The values for the parameters are specified either by settings file or program input argument. The priority of input arguments is higher than input settings file. User can define values of parameters which are not frequently changed in settings file and the remaining from program arguments. In this case specifying setting file is mandatory. If the settings file is not given, user has to define values for all parameters in arguments list. Program accepts arguments in any order.

The program expects to find the following variables in the settings file:

- **JLib**: This is the path of all Java standard libraries.
- **ProgLib**: This variable specifies a set of classes supposed to be analyzed.
- **OLib**: If there is third party library used by the program, can be defined using this variable.
- **EClass**: The value of the variable specifies entry class of the program. Entry class is the one contains entry method of the program.
- **EMethod**: The entry method of the program is determined by this variable. The entry method must be in entry class.

The setting file must have values for all five variables. None must left without value. Corresponding switches for these variables are as follows:

- **JLib**: `-java`.
- **ProgLib**: `-prg`.
- **OLib**: `-lib`.
- **EClass**: `-entc`.
- **EMethod**: `-entm`.

To report the result control flow graph, `-out` along with output file name can be used. If output file is not defined, the result is printed in console.

### 5.4 Modular CFG Extractor

Modular CFG extractor (ModFGExtractor) does not need to have knowledge about complete program. Like ComFGExtractor, the process starts by loading required settings. Then specified set of classes (sub-program) is parsed. Our working set will be all concrete methods in all sub-program’s classes. CFG construction rules are applied for each method defined inside a class. General picture of the program, its usage, output formats and starting point of the program are similar to
ComFGExtractor. So in following we will explain functions which are specific to ModFGExtractor.

**Parsing Sub-Programs**

Function `parse_subprogram` is to parse a path of an open system or (sub-)program and to build the nodes with hierarchical connections. Given a path and a list of names it starts to load class files. Then using `add_inc_classes` builds the nodes of the (sub-)program. Finally `build_hierarchy` is employed to establish generalization relationships.

Function `add_inc_classes` given a set of raw and disjoint classes, tries to find the relationships between classes and make a (sub-program). For each member of class list, `add_one_node` is called to make a place for the given class `cs` in the (sub-)program. If the type is `Class` then `add_classFile` is responsible to build and set the node. If the type of class file is an Interface then `update_interfaces` is used to place the built node. Currently we assume there is no interface and if a type of interface is found in loaded files it is ignored.

Function `build_hierarchy` parse the whole entities in a (sub-)program and discovers hierarchical information between them. Since we are considering open systems here and there is a possibility for missing elements, if a class claims about having a parent but the node of the parent is not found, None is assigned as the type of its super-class. So there might be inconsistency between classes hierarchical raw information and relations built between nodes in (sub-)program.

**CFG Extractor**

Function `get_prg_cfg` extracts CFG from a given (sub-)program. The result CFG is disjoint union of all CFG of classes in `prg`. For each class all concrete methods are listed, then `const_cfg` starts to build the CFG starting from first address of the method’s body.

Function `const_cfg` is responsible for building the CFG of a method based on the construction rules. For each instruction corresponding rule is applied to establish an edge between two nodes (program points). This function also collects all instantiated classes to be used in RTA based virtual method call resolution. Virtual calls is approximated with CHA algorithm in this function.

**Virtual Calls**

Function `resolve_vmc` tries to resolve virtual call in a given environment `prg`. Currently this function is implemented for algorithms CHA and RTA. If the given algorithm is CHA then `cha_vmc_resolve` is used to find possible receivers. If algorithm is RTA then intersection between instantiated classes and results from CHA will be final possible receivers. This function is extensible to define more virtual call resolution algorithms. In fact it plays an interface role between CFG construction engine and different virtual call resolution algorithms.
Using CHA algorithm virtual calls can be accomplished while we are constructing CFG, because what CHA needs can be obtained by structural information. By the time of CFG construction (sub-)program contains hierarchy information. But in order to resolve virtual call using RTA, we need to have information about instantiated classes. So RTA resolution should be postponed to have complete parse of all instructions. In current implementation CHA is used as a safe approximation of possible receivers. Virtual calls can be pruned in final step of CFG construction using instantiated classes set. Another, maybe not very efficient, solution could be to scan all instructions of methods and collect instantiated classes before CFG construction. Then we can use this set to resolve virtual calls using RTA.

Function cha_vmc_resolve given class $c$ and method signature $ms$ in environment $prg$ finds classes which are sub-types of $c$ and override method $ms$. Actually $c$ is static type of the object being called in a virtual method call instruction. This function at first uses get_subtypes to get the list of all classes in $prg$ which are sub-types of $c$. Then in a recursive function get_possible_rcvs checks if given method signature $ms$ is a member of class methods list. If the class contains a method with signature equal to $ms$ then this class is considered as a possible receiver. We need to note here that static type of object being called is one of the possible receiver and last statement of the function ensures this fact.
Chapter 6

Conclusions: Static Analysis

6.1 Related Work

Much work has been done in control flow analysis of programs in different programming languages, mostly aiming at data-flow analysis, static and dynamic slicing of programs and especially different kinds of program testing. Allen [6] is one of the pioneers. To introduce just a few of these works [33, 34, 35, 36] are cited.

Based on the application of the control flow analysis a variety of representations has been proposed. Call graphs, control dependency graphs, data dependency graphs and control flow graphs are some well-known graph based models constructed in control flow analysis. To our best knowledge, this study is the first work on provably sound control flow graph extraction of Java byte-code programs.

Harrold et. al. [37] propose algorithms to construct control dependency graph of the program while the source code is being parsed. During parsing of the program control flow information is calculated and it is incorporated in program dependence graphs (PDG)\(^1\) construction. Thus results in an efficient PDG construction without requirement for any pre-computations. The presented algorithm is developed for programs in the C language.

Sinha et. al. [40, 39] propose criteria for testing exception handling constructs in Java programs (Java source code). In their research they consider the effect of exception propagation and exceptions type conversion. Intra-procedural analysis (individual procedures) builds \texttt{throw-catch-finally} constructs of the methods and inter-procedural analysis (interacting procedures) builds control flows between the methods. Abstract syntax tree (AST) of the Java program is parsed to extract CFG. The proposed algorithm for CFG construction traverses the AST of the program and then inter-procedural CFG (ICFG) is established. In ICFG construction four types of edges are constructed between procedures: call edges, return edges, exceptional-return edges and unconditional-transfer edges. Normal CFG is

\(^1\)In program dependence graph (PDG) [38] nodes represent statements and edges represent control or data dependencies. It encodes both control and data flow information in the result graph.
constructed using algorithms proposed in [41, 42].

In a similar work Jiang [43] propose an algorithm to extract exception control flow graph (ECFG) of C++ programs. In the proposed model of the programs implicit control flow of exceptions and exceptions propagation is represented. Based on the inter-procedural ECFG (ICFG) they described techniques for path testing and definition-use testing of C++ programs.

Jo and Chang [44] propose a method to construct CFG by computing separately normal flow and exception flow of Java programs (Java source code). Using a set-constraints of exceptions and iterative fix-point method they compute exception propagation paths. Set-constraints rules are defined for statements throw, try-catch, MethCall, MethDecl. Finally they show that CFG of a program can be constructed by merging an exception flow graph onto a normal flow graph.

Cousot and Cousot [22] study modular program analysis by abstract interpretation. Compositional static analysis of the programs are categorized in different groups, studied in chapter 4. The study considers a broad range of works in modular program analysis.

Probst [21] aims to prevent re-analyzing of libraries and proposes a technique so that libraries are analyzed only once. Then the result of the analysis is saved as a constraint graph. So during the analysis of the program the graph for the methods to be analyzed are selected and there is no need to repeat analysis. The exceptions are not reported in this study. Class hierarchy information and class declarations (which method belongs to which class) are assumed to be given.

Ganaim and Spoto in [15] present information flow analysis of Java byte-code programs based on the denotational semantics of instructions. As a result Spoto introduced Julia [16] as a generic static analysis tool developed in Java programming language for Java byte-code programs. Optimization and verification is the objective of this tool. Static analysis is performed through denotational or constraint-based fixpoint calculation. We found this work as the only formal study using programs semantics.

A program analyzer developed using Soot framework [4] was being used in the verification tool set [1, 2, 3]. Soot, is a framework developed in Java programming language for optimizing and analyzing of Java byte-code. Soot is based on several kinds of intermediate code representations. At first step Soot translates the Java byte-code into one of these intermediate languages to make the static analysis simpler. Then in the second phase analysis starts based on these intermediate translation of the programs.

Whaley and Lam [24] are using the idea of method’s summary graph to have an efficient flow sensitive algorithm for points-to analysis of Java programs. In this algorithm semantics of methods are represented by a precise sparse summary graph. Then the algorithm iterates over methods’ summary graphs to reach a fixed point solution. Utilizing the access paths and field information in the summary graphs helps to have a quick and effective analyzer for large programs.
6.2 Future Work

The correctness of the compositional CFG extraction for open systems is an important future theoretical task. Based on the CFG behavior for open systems and the CFGs composition operator the soundness of the CFG extracted using modular methods can be proved.

Stack analysis to capture the type of exception in instruction `throw` can be removed in CFG construction if the program is transformed to a stack-less intermediate language. BIR can be considered as the only provably correct intermediate language for Java byte-code [32]. However, still there is no study reported for exceptional handling of programs. Re-defining of the rules for BIR instruction set makes CFG construction process more easy and efficient. The first step will be JBC program transformation into BIR, which is available thanks to Sawja modules. The intermediate program will be free from stack based calculations instructions, which are not important for our CFG. This will result in a graph with fewer elements. Then in the second step CFG construction rules can be applied to build program’s control flow graph. The final step will calculate possible receivers in virtual method calls and exception propagation through method invocations edges.

Symbolic execution of methods and building method or class summary graphs are a strongly recommended task in modular CFG extraction. These techniques provide a powerful and efficient semantics for local analysis of each module. Symbolic execution of methods provides a flow-sensitive procedure to find possible receivers in virtual calls. To find fewer candidates in virtual calls will result in fewer call edges, thus accelerate the process of exception propagation through call edges.

User provided interfaces of abstract modules are needed to be expressed and analyzed in modular CFG extraction of open systems. One of the main future tasks is to express the semantics of the abstract modules using a proper formal language. Then the program needs to analyze and extract required information of these interfaces and build the global CFG in collaboration with the results from the analysis of concrete modules.

Usually design models of modules are available in early stages of components development. Also there is no need for additional education in interface definitions. Control flow analysis based on design models of abstract modules can be considered as a novel approach in compositional verification tool set and user provided modular CFG extraction. UML2.0 provides all required method execution control information like loops, conditional statements and branches, which are beneficial for CFG extraction.

Method reachability analysis is considered as a task to improve RTA based approximations. Currently we collect class instantiations inside methods bodies without analyzing the possibility of execution of the methods. So this leads to a variant of RTA algorithm with more possible candidates for virtual call resolutions.

At the implementation level, exceptional edges filtering can be useful to decrease the exceptional edges which are not common in programs. Currently the rules capture all the possible exceptions in the instructions.
Control point simplification can be an implementation level task in the current version of the tool. In the current version of the CFG extractor we consider all edges regardless of the edge labels. We could merge control points with one outgoing edge. This would result in a smaller graph. But in that case the graph rules would not be upgradeable to program models which require data analysis. There is a plan to extend the current compositional verification method to boolean programs, so edges with computational instructions will be required.

Unification the CFG extractor with the verification tool set is an implementation level task which helps the whole process of program model extraction and verification can be accomplished in one phase. Currently the verification tool set reads the extracted CFG from a file, which is the output of the CFG extractor.

6.3 Summary

In the control flow graph extraction techniques for Java byte-code programs using static code analysis, we proposed rules for CFG construction of programs. Then we proved that the resulting CFG is sound, meaning that it simulates all possible behaviors of the corresponding code. Based on the basic rules we studied possible techniques in order to extract CFG of open systems, where some of the modules are not available through their implementations. CFG extractor for closed systems was implemented using Javalib and Sawja framework in OCaml language.

Soot based program analyzer has been replaced by the version of the program developed for closed programs. All the possible errors and bugs have been fixed during experiments in compositional verification tool set.

The basic building blocks for modular CFG extraction have been developed and still need some effort to be finalized. A hybrid approach based on worst-case, user provided and symbolic execution is recommended as the main approach for modular CFG extraction. The first phase could be a worst-case CFG extraction. Then based on the interfaces for abstract modules it can be upgraded. Symbolic execution and module summaries are suggested as final phase to improve the efficiency.

In this study we defined formal rules based on program instructions. But algorithms on how to calculate the elements are not considered. Exceptions propagation and possible call edges in modular CFG extraction are main procedures demanding algorithms to find the fixed-points. Standard iterative algorithms to find the fixed-points are applicable in all these procedures.
Part III

Graph Transformation
Chapter 7

Graph Transformation

Graph Transformation is our second approach to extract control flow graphs from Java source code. In the following chapter we introduce this technique and some basic concepts as background.

7.1 Introduction

In system modeling and complex structures description languages, graph is a well-known, well-understood and frequently used concept to model system in a direct and intuitive way. Simplicity and power of graph gives an approach to a wide range of problems, almost everywhere in computer science and software engineering. Data and flow diagrams, the structure of an object-oriented system, modeling the static and dynamic aspects of systems and many other concepts can be viewed as a graph.

Graph grammars and graph transformation is an area of computer science, in which established methods, techniques and tools are widely used. The idea of graph transformation relates to techniques of generating a defined target graph from a known source graph by applying re-writing rules. In general graph transformation techniques consist of three major components: source graph, transformation rules and target graph. Rule-based modification of graphs is another term used for graph transformation. Program transformation, program analysis and verification are active areas in graph transformation techniques.

In this work we are going to take advantages of the research results obtained in developing Groove [46] \(^1\). We define rules for virtual method calls resolution using algorithms CHA and RTA. Then we use abstract syntax graph of the program and apply our defined rules to extract final flow graphs based on our program model. To start with, we provide some background required in the next chapter, where our rules will be presented.

\(^1\)Groove (GRaphs for Object-Oriented VErification) is a graph transformation tool set developed in FMT group at University of Twente.
CHAPTER 7. GRAPH TRANSFORMATION

7.2 Background

In general graphs are defined as a pair of two sets: a set of nodes and a set of edges. In directed graphs, an edge is an ordered pair of nodes while in an undirected graph an unordered pair of nodes identifies the edge. Definition 18 gives a variant definition for directed graphs which is usually used in graph transformation context.

**Definition 18 (Graph [47])** A graph \( G = (V, E, s, t) \) consists of a set \( V \) of nodes (also called vertices), a set \( E \subseteq V \times V \) of edges, and two functions \( s, t : E \to V \), the source and target functions.

Generally, in the literature, a graph \( G \) is defined as: \( G = (V, E) \) in which \( V \) is set of nodes and \( E \subseteq V \times V \) denotes set of edges. This definition is an equivalent presentation with the one defined in Definition 18, for an element \( (v, w) \in E \), \( v \) represents its source and \( w \) its target node, but using functions for identifying source and target in parallel edges makes it more convenient to express.

Graphs are employed here to model the syntax and semantics of program structure. So we need to make a unique mapping between graphs and program elements written in a programming language, say Java. Definition 19 introduces labeled graphs which will be used to mark nodes and edges with proper programming language elements.

**Definition 19 (Labeled Graph)** A label alphabet \( L = (L_V, L_E) \) consists of a set \( L_V \) of node labels and a set \( L_E \) of edge labels. A labeled graph \( G_l = (V, E, s, t, l_V, l_E) \) consists of an underlying graph \( G = (V, E, s, t) \) together with labeling functions \( l_V : V \to l_V \) and \( l_E : E \to l_E \).

A transformation rule is defined as \( p = (L, R) \), in which pair of graphs \( (L, R) \), makes the core of the rule. Now if we have a source graph \( S \), then applying of rule \( p = (L, R) \) means finding a match of sub-graph \( L \) in \( S \) (\( S \) contains subgraph \( L \)) and replacing it with \( R \) leading to target graph, \( T \) (figure 7.1).

![Figure 7.1. Rule based modification of graphs](image)

Graph transformation can be used to model computations. If one state of system can be modeled as a source graph, then every computation step will be an
7.2. BACKGROUND

application of a rule, resulting in the target graph, which is the next state of the system.

As mentioned earlier, applicability of a graph transformation rule, mainly relies on pattern matching in the host graph. Definition 20 defines the concept of matching in graphs.

**Definition 20 (Graph Matching)** Let $G$ and $H$ be labeled graphs. A matching of $G$ in $H$ is given by a function $m : V_G \rightarrow V_H$, such that: $e = (v, w) \in E_G$ and $l_V^G(v), l_V^G(w) \in L_V^G$ and $l_E^G(e) \in L_E^G$ implies $v' = m(v), w' = m(w) \in V_H$ and $e' = (v', w') \in E_H$ and $l_V^H(v') = l_V^G(v), l_V^H(w') = l_V^G(w) \in L_V^H$ and $l_E^H(e') = l_E^G(e) \in L_E^H$.

Now based on the concepts defined for labeled graphs and matching we are ready to define the production or rule. Intuitively each production is a graph rewriting rule which if applicable, results the target graph. So let define a production in Definition 21.

**Definition 21 (Production)** A production rule $p$ is a tuple $(L, R)$ of graphs, which $L$ and $R$ are known as left- and right- hand side graph, respectively. A rule is applicable to source graph $G$, if there is matching of $L$ in $G$. A set of production rules is often called a production system.

To define the application of a rule $p = (L, R)$ on host graph $G$, the sets of elements to be deleted, preserved and created should be expressed. Generally speaking the derived graph $H$ is constructed by removing those elements which are in $L$ and not in $R$, then adding elements which are in $R$ and not in $L$. Simply we can formulate this production as

$$H = G - (L - R) \cup (R - L)$$

Example 7 shows how a production is applied to a host graph and generates the target graph. Actually this is a simple example of decorating a while loop with flow information using graph rule.

**Example 7 (Rule Application)** Figure 7.2 illustrates a graph transformation rule applied on source graph $S = (V, E, s, t, l_V, l_E)$ in which

- $V = \{v_1, v_2, v_3\}$, $E = \{e_1, e_2\}$,
- $s = \{(e_1, v_1)(e_2, v_1)\}$, $t = \{(e_1, v_2)(e_2, v_3)\}$,
- $l_V = \{(v_1, \text{While}), (v_2, \text{Condition}), (v_3, \text{Block})\}$, $l_E = \{(e_1, \text{condition}), (e_2, \text{block})\}$

and the rule is $p = (L, R)$ where elements of $L$ is indicated with del: and new: is used to mark the elements of $R$. The result graph shows flow between elements of source graph.
In some cases application of rules may be problematic and cause some complicated situations. **Matching conflicts** and **dangling edges** are two main problems in rules application. Matching conflict refers to a situation in which more than one matches are found in source graph for $L$. Dangling edge refers to a situation in which the source or target node of an edge is removed but not the edge.

The way to handle these problematic cases is resulted in two approaches in algebraic graph transformation: Double-PushOut (DPO) and Single-PushOut (SPO). DPO does not allow applying these problematic rules, but on the contrary SPO gives permission to such direction with giving priority to deletion over preservation. Here we are not studying details of these techniques. For a detailed and theoretical discussions about approaches [47] is suggested.

### 7.3 GROOVE

Groove [46] is a tool developed with the aim of Java programs static analysis and verification using graph transformation techniques. Groove can be considered as a tool for state space exploration where states are represented as graphs and transitions from one state to another state are given by graph transformation rules. For a number of case studies accomplished using Groove and a comparison with other graph transformation tools [48] is suggested.

Groove takes advantage of abstract syntax tree (AST) generated by Java compiler and represents the program as an abstract syntax graph (ASG), enhanced version of the AST with variables binding information. The goal of program syntax tree is not to represent the graphical model of the program. Syntax graph is used as an initial data structure to reach designed purposes in Groove.
7.3. GROOVE

Groove is capable of defining some constraints on nodes and edges combination by specifying a *type graph*. There is also possibility to work in un-typed mode. In un-typed mode all graphs (source, targets and rules) can be in any arbitrary combinations, but in typed mode all must comply with specific predefined typed graph(s). [49] defines a type graph as follows:

**Definition 22 (Type Graph)** A type graph is a labeled graph with the following additional structure:

1. **Inheritance**: There is a binary acyclic inheritance relation over the nodes. The idea is that the node type of the source of each individual edge instance is smaller or equal, according to this inheritance relation, than the source of its edge type;

2. **Composition**: A subset of the edge types are marked as composition edges. Such edges encode a part-of relation: the target node instances are considered to be part of the source node instances with which they are connected. Because a node cannot be part of more than one other node, the composition edge instances should form a tree within the instance graph.

3. **Ordering**: A subset of the composition edge types are marked as ordered. For all such edge types, there should be a total ordering among the target node instances that are connected by these edge instances to (and hence are part of) any given source node instance. This ordering will be encoded by adding integer index attributes to the target nodes, which contain the sequence number within the ordered list.

4. **Multiplicity**: Every edge has an associated multiplicity, which is a range \(i..j\) of natural numbers with \(i < j\), or such that \(j = \ast\). The multiplicity indicates how many edge instances there should exist for every individual source node instance, where \(\ast\) stands for an unbounded number.

*Graphical notations for type graph* is mostly followed by UML [50] in Groove.

Groove provides two type graphs: *ast-type* and *fg-type*. Constructing abstract syntax graph of the program, Groove employs *ast-type* to control constraints in variable binding rules. Then flow decoration production system takes advantages of *fg-type* type graph.

Groove transforms the graph by applying the applicable rules in source. We know from graph rule definition that a rule specifies the elements which should be preserved, removed and created in the host graph. Groove uses graphical notations and colors to describe a rule.

In Groove the elements to be preserved are depicted in black with continuous line. Elements colored in blue with dashed thin picture represent the pattern which must be detected in host and deleted in target. Green continuous fat elements are creators, meaning that they are not in LHS but applying the rule, they will...
be appeared in RHS. Finally, elements in red with dashed fat presentation, show a pattern that its presence in host graph prevents the rule from being applied. Groove names preserved elements as Readers, deleted elements as Erasers, new elements as Creators and constraints as Embargoes [51].

Groove employs two control mechanisms to supervise the sequence of applications. The simplest and common way is to assign priority to each rule. In rules application process the first applicable rule with highest priority is selected. Groove also equipped with a control language for rule applications. The control language makes it possible to define any arbitrary sequence of rule applications. Looping, random selection, conditional selection and simple (non-recursive) function calls are the important features of the language that can be useful. More detail information about the language is available in [51].
Chapter 8

CFG Extraction using Groove

After the prerequisites of the graph transformation technique in program analysis have been presented, we explain the rules defined for control flow graph extraction. This chapter presents the rules developed to extract CFG of methods declared in a Java class.

8.1 Introduction

In the previous chapter we have learned that Groove utilizes the artifact of the graph compiler, which is the abstract syntax tree (AST) of the program. Then applying variable binding and the control flow production system, developed in [49], we obtain the abstract syntax graph (ASG) of the program decorated with flow information in final state. Our aim is to extract the program control flow graph based on our program model. Virtual method call resolution is our main challenge in the development of the rules. Figure 8.1 depicts the chain of actions we have to take to extract the control flow graph from a given Java program.

In the following we explain the rules designed for basic control flow graph extraction. Program structures like conditional statements, loops, sequential statements will be described here. Then based on the patterns we have in the source graph we design rules for virtual method call resolution. The rules here are designed in the Groove tool set. We will not explain how to install or how to work with Groove. The tool set documentation [52] is suggested for this purpose.

8.2 A Production System for CFG Extraction

Referring to the program model defined in modular verification tool set, we can see that target CFG consists of nodes as program control points connected via edges labeled as $\epsilon$ (empty transition) for computational transitions and call for transitions in which a method call happens. Entry node has property entry and return points’ property will be ret. In order to explain designed graph grammars for CFG extraction we classify them in two different categories:
8.2.1 Basic CFG

The final state of the production system designed for control flow information of the program contains all flow edges between source code statements. Thanks to type graphs and using generalization concepts we are able to extract our required CFG from the result of control flow production system application. ASG generated by `java-control-flow-build.gps` production system, contains `FlowElem`s connected by edges that a sub-set of the edges carries program flow information. So if we can remove all un-necessary edges which has no useful flow information and generalize all nodes as a `FlowElem`, then fundamental CFG is obtained. So let start with type graph defined for our designed production system.

Type Graph

First we define our type graph to express the relationships of elements in target graph. Using conceptual generalization on source graph (ASG) and target graph, we can build type `Flow` which is specialized to `FlowNode` and `FlowEdge`. Due to type `ast-type` and `fg-type` type graphs, every abstract syntax node is a kind-of `FlowElem`. Figure 8.2 shows the defined type graph in Groove that expresses the
constraints of all states of target graph generation using production system. Each flow element in source graph is associated to our generalized node \textit{Flow}. \textit{FlowNode} and \textit{FlowEdge} are two sub-types of \textit{Flow}. Each flow transition in source graph will be transformed to one of the members in \{empty, to, out\}.

![Diagram](image)

\textbf{Figure 8.2.} Target control flow graph Type Graph

In our type graph, the source and target of every \textit{empty} transition is \textit{FlowNode}, the source of \textit{out} transition is \textit{FlowNode} and the target terminates in \textit{FlowEdge}. The opposite direction is established for \textit{to} transition. Actually \textit{FlowEdge} is designed to indicate method calls in final CFG.

Every \textit{FlowNode} contains a set of properties and in our type graph \textit{FlowProp} connected with \textit{prop} to \textit{FlowNode} is responsible for expressing properties of control points of the program. As we know from labeling function of a CFG each control point is marked with corresponding methods signature and every control point label should indicate that whether the control point is an exceptional or a normal control point of the program. Two \textit{MethodProp} and \textit{ExceptionProp} of the type graph are designed to express two types of general properties that a control point can express. Attributes defined in \textit{FlowEdge} will be explained later, where virtual method call resolution will be discussed.

\textbf{Starting Point}

Each method in the code corresponds to a unique control flow graph. The first step in CFG extraction is to build the entry point of the method’s CFG. Rule named \textit{MethodDecl} detects the node type \textit{MethodDecl} and establishes related \textit{FlowNode} connected with edge labeled \textit{target}, if it does not have any previously constructed \textit{FlowNode}. The constraint is defined because each method declaration relates to a unique \textit{FlowNode} and to prevent infinite rule application. Method name alongside with class name will be property of all flow nodes. Figure 8.3 shows the rule to find the starting point of a method declared in the class.
Graph Traverse

Having found the entry point of the method’s CFG we need to traverse elements following flow edges. Flag built is used to control the sequence of the FlowNode construction and rule application. Having constructed the FlowNode related to each FlowElem, the flag build is propagated through source graph nodes applying build-propag rule. Figures 8.4 shows the propagation rule for flag build in graphical rule definition notation.

Construction

Detecting the first entry point of the flow graph, we need a rule to generate a FlowNode for each FlowElem and establish our program model’s transitions. Rule named Rule-FlowNode, finding the pattern of built FlowElem with corresponding FlowNode, if the flow sequence has next FlowElem, creates the target FlowNode connected to the previous step with empty transition. This transition labeled empty will be computational transition on our program model. The constraints express that:

- The FlowElem is currently being built must not be of type MessageSend, because for message send, FlowEdge is created.
8.2. A PRODUCTION SYSTEM FOR CFG EXTRACTION

- The FlowElem built in previous step must not be of type ReturnStmt, because return statement is supposed to be the last node with property ret.  

- Since FlowNode is unique for every FlowElem, the FlowElem under construction is not supposed to have another FlowNode.

Constructed FlowNode uses previously built flow nodes property, which is shared among all flow nodes. In final step of FlowNode creation, corresponding FlowElem is tagged with flag built to identify that the task for the node is accomplished. Figure 8.5 depicts the graphic representation of what we have described of Rule-FlowNode.

**Figure 8.5. Flow Node Construction Rule**

Joint Points

Although Rule-FlowNode and build-propag rules are enough to traverse all sequential flow elements on the source graph connected with the flow transitions, but in multi-paths joined with a flow element, the joint node is marked with flag built via one of the paths earlier than the other path(s). Therefore, it results in two adjacent flow elements with flag built (figure 8.6) while the flow edge between the corresponding flow nodes is not established yet. As a consequence the target graph contains two disconnected FlowNodes.

Therefore we need an extra rule for this situation to connect two flow nodes and prevent dis-connectivity between two built flow nodes in target CFG. The situation happens because there is no control on rule application with multiple paths joint in a flow element. So rule FlowNode-2 (figure 8.7) is defined simply to connect two flow nodes if their related flow elements in host graph has flow transition and there is no empty connectivity between them.

---

1 The source ASG decorated with flow information contains more transitions facing return statement: abort and exit leading to node Exit.
Final Step

Any flow graph is terminated when a return statement is found. Final defined rule is `ReturnStmt`, which adds the property `return` if the flow element related to the flow node has type `ReturnStmt`. Figure 8.8 shows the definition.

8.2.2 Method Calls

So far we have defined rules for basic flow structure construction of the programs, which are able to generate desired output consisting of flow nodes connected with empty transitions. Target CFG in our program model discerns between method call and computational transitions. So it is imperative to have distinct rules to create required flow information in case a method call appears in the program. As we remember in flow node construction rules we have defined a constraint on flow element that if the element is of type `MessageSend` then the rule is not applicable.
8.3. VIRTUAL METHOD CALL RESOLUTION RULES

Considering type graph definition \( \text{FlowEdge} \) is a node defined to represent a method invocation. It can have to and out as an ingoing or outgoing edges. Rule MethCall-1 defines how to behave when a message send happens in the way of flow path. When the build propagation rule, indicates the \( \text{MessageSend} \) as an active flow element, then a node of type \( \text{FlowEdge} \) is created in the target graph with an edge, labeled as out, connected to preceding \( \text{FlowNode} \) (figure 8.9). The signature identified by \( \text{MessageSend} \) is copied as a property of \( \text{FlowEdge} \). Similarly rule MethCall-2 constructs the ongoing flow sequence of method call (figure 8.10).

![Figure 8.9. Method Call Rule 1](image)

![Figure 8.10. Method Call Rule 2](image)

Like joint points for flow nodes we need another extra rule to cover joint points in which \( \text{FlowEdge} \) is the joint point in multi-path flow sequences. Figure 8.11 shows the rule defined for cases in which flow edge is located in a joint point.

8.3 Virtual Method Call Resolution Rules

As explained earlier, there are various algorithms designed in the area of virtual method call resolution and points-to analysis of Java programs. To have a safe approximate control flow graph, CFG extractor needs to exploit virtual method resolution graph rewriting rules based on program static analysis in its production system. Here CHA and RTA are our chosen algorithms as two rudimentary algorithms that we designed rules to use them to find possible receivers in virtual
method calls. Algorithms are explained in background chapter and here we just present graph re-writing rules together with examples.

8.3.1 Class Hierarchy Analysis - CHA

Now we are ready to define CHA graph transformation rules based on the algorithm definition. CHA rules need to have hierarchy information for the set of classes. Each type is represented in program graph as a node with type `TypeDecl`. If two types has `inheritance` relation, the sequence of relations is: `TypeDecl` has `superClass` relation with `TypeRef` which has `refersTo` relation with parent `TypeDecl`. This pattern is detected in the source graph and an edge labeled `subType` is established from child to parent node.

Let us assume `Type` as our set of class types in the program being analyzed and `T ∈ Type` be representative of node `TypeDecl` in ASG. Relation `SubType ⊆ P(Type × Type)` is constructed using following rules:

- **SubType-1**: which is applied when `(T_i SubType T_j)` or `(T_j SuperClass T_i)`
- **SubType-2**: `(T_i SubType T_j) ∧ (T_j SubType T_k) ⇒ (T_i SubType T_k)`

Intuitively rule `SubType-1` constructs sub-typing edge for each pair of `TypeDecl` with `inheritance` relation. Rule `SubType-2` applies the transitivity property of the relation. Figures 8.12(a) and 8.12(b) shows the rules defined in Groove for sub-typing.

Second set of CHA rules implements the `staticLookup(T, m)` and creates an edge labeled `CHA-bind` between `MessageSend` and `TypeDecl`, if methods have the same signature, expressing possible target types for method call. Rules containing node `LocalDecl` covers code fragments including call sites that object variable is declared as local variable (figures 8.14 and 8.14), while `FieldDecl` indicates object is an attribute declared in the class (figures 8.15 and 8.16).
8.3. VIRTUAL METHOD CALL RESOLUTION RULES

So far we have made the links from call site to all possible types. In final step rule CHA, is defined to collect all bound types in target control flow graph. CHA accumulates all TypeDecl names of which method call is bound (CHA-bindsTo), as property values of FlowEdge, named CHA-targetType (figure 8.17).

Possible receivers of the message in method call are based on the sub-typing relations between types. Therefore we need to put highest priority for sub-typing rules, then place the next level of priority for CHA rules. In this way first all hierarchy information is discovered in the source graph and then they are used to bind call site object to the possible type declarations.

8.3.2 Rapid Type Analysis - RTA

RTA production system relies on the final state of CHA production system. Based on the RTA definition, the algorithm prunes all types not instantiated in the program from the CHA collected possible types. At first step we need to detect the class instantiation expression in source graph. Then we can consider instantiated type as a possible receiver in RTA solution set, if the type is a member of CHA possible receivers set.
CHAPTER 8. CFG EXTRACTION USING GROOVE

Figure 8.14. Rule-CHA-L1

Figure 8.15. Rule-CHA-F1

Figure 8.16. Rule-CHA-F2
8.3. VIRTUAL METHOD CALL RESOLUTION RULES

Rule RTA-B detects the pattern of allocation expression, which is node AllocExpr. The node must be connected to the type reference of the object supposed to be instantiated. Then if the resolved type of allocation expression is equal to one of the CHA-bound type declarations, then RTA-bindsTo relation is established between MessageSend and TypeDeclaration. Finally in control flow graph extraction process, rule RTA-T acts on FlowEdge to make a list of RTA reachable types. Figures 8.18 and 8.19 depicts the rules in graphical notations.

We need to define the priority for CHA higher than RTA rules. In this way first all hierarchy information is discovered in the source graph and then they are used to bind call site object to the possible type declaration. This results in possible CHA resolved set. Then if there is no more applicable rule from CHA algorithm, in
the next level, rules from RTA will start to find the instantiations and collect RTA possible types.
Chapter 9

Conclusions: Graph Transformation

9.1 Related Work

H. Kastenberg et. al. in [56] developed a small object-oriented language which additionally to imperative constructs also includes inheritance and method overriding. In this work they defined the semantics of the language graphs, in terms of the graph production system. The execution of the program is simulated by transformation of a given program into the graph and applying the production system.

Ruben Smelik et. al. in [53] proposed a visual, graph based specification language for specifying the control flow semantics of programming languages that builds upon the syntax of the language. This specification language, called CFSL, gives the possibility to the language designer to write the specification. A graph production system to construct control flow graphs is translated from CFSL. Given BNF rules of the language, an abstract syntax graph (ASG) is generated. CFSL acts as a type graph and control flow production system decorates the program with flow information. The CFSL language is validated on the Java programming language in [54].

Sombekke in his thesis [55] used graph transformation techniques to specify the semantics of .Net Intermediate Languages (IL). In this project byte-code of ILs is translated as a start graph to apply graph production systems. Designed production system simulates the execution of the original IL. Control flow information is not presented explicitly in this translation and production system employs labels and semantics of the instruction to to simulates the execution.

9.2 Future Work

In order to have a complete set of graph grammars for compositional verification with CVPP [3], exceptional control flow graph extraction rules require to be developed in the future. Detection of try-catch-finally statements and analyzing of the statements with possible exception set will be the main task.

In the graph transformation approach we have not addressed to the modular
control flow graph extraction of Java programs. The main step toward this goal is using a modular compiler to compile an incomplete program. Then Groove needs to be able to load modules of the programs independently. The CFG extraction rules defined here are applicable but we still need rules for global control flow analysis.

The compositional CFG extraction methods discussed in chapter 4 can be employed here to equip the Groove with modular CFG extraction. Modular graph production system to extract the CFG of an open system requires a practical method to analyze abstract components. If a component is not available through its implementation, we can take advantages of user-provided interfaces. To achieve this, the Groove needs to have the ability of analyzing components specification and extract required information for global analysis.
Bibliography


Programming Systems, Languages, and Applications (OOPSLA’96), SIGPLAN, 1996.


