Sound Modular Extraction of Control Flow Graphs from Java Bytecode

PEDRO DE CARVALHO GOMES

Licentiate Thesis
Stockholm, Sweden, 2012
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i datalogi onsdagen den 12 december 2012 klockan 13.00 i sal D3 Lindstedtsvägen 5 Kungliga Tekniska högskolan, Stockholm.

© Pedro de Carvalho Gomes, November 16, 2012

Tryck: E-print
Abstract

Control flow graphs (CFGs) are abstract program models that preserve the control flow information. They have been widely utilized for many static analyses in the past decades. Unfortunately, previous studies about the CFG construction from modern languages, such as Java, have either neglected advanced features that influence the control flow, or do not provide a correctness argument. This is a bearable issue for some program analyses, but not for formal methods, where the soundness of CFGs is a mandatory condition for the verification of safety-critical properties. Moreover, when developing open systems, i.e., systems in which at least one component is missing, one may want to extract CFGs to verify the available components. Soundness is even harder to achieve in this scenario, because of the unknown inter-dependencies involving missing components.

In this work we present two variants of a CFG extraction algorithm from Java bytecode considering precise exceptional flow, which are sound w.r.t to the JVM behavior. The first algorithm extracts CFGs from fully-provided (closed) programs only. It proceeds in two phases. Initially the Java bytecode is translated into a stack-less intermediate representation named BIR, which provides explicit representation of exceptions, and is more compact than the original bytecode. Next, we define the transformation from BIR to CFGs, which, among other features, considers the propagation of uncaught exceptions within method calls. We then establish its correctness: the behavior of the extracted CFGs is shown to be a sound over-approximation of the behavior of the original programs. Thus, temporal safety properties that hold for the CFGs also hold for the program. We prove this by suitably combining the properties of the two transformations with those of a previous idealized CFG extraction algorithm, whose correctness has been proven directly.

The second variant of the algorithm is defined for open systems. We generalize the extraction algorithm for closed systems for a modular set-up, and resolve inter-dependencies involving missing components by using user-provided interfaces. We establish its correctness by defining a refinement relation between open systems, which constrains the instantiation of missing components. We prove that if the relation holds, then the CFGs extracted from the components of the original open system are sound over-approximations of the CFGs for the same components in the refined system. Thus, temporal safety properties that hold for an open system also hold for closed systems that refine it.

We have implemented both algorithms as the ConFlEx tool. It uses SawJa, an external library for the static analysis of Java bytecode, to transform bytecode into BIR, and to resolve virtual method calls. We have extended SawJa to support open systems, and improved its exception type analysis. Experimental results have shown that the algorithm for closed systems generates more precise CFGs than the modular algorithm. This was expected, due to the heavy over-approximations the latter has to perform to be sound. Also, both algorithms are linear in the number of bytecode instructions. Therefore, ConFlEx is efficient for the extraction of CFGs from either open, or closed Java bytecode programs.
Contents

Acknowledgements

1 Introduction
   1.1 Contribution
   1.2 Organization

2 Background
   2.1 Formal Java Virtual Machine Framework
   2.2 Program Models
   2.3 Direct Extraction of CFGs from Bytecode
   2.4 The BIR Language
   2.5 Compositional Verification

3 Extraction of CFGs from Closed Systems
   3.1 Transformation from BIR into Control Flow Graphs
   3.2 Correctness of CFG Extraction

4 Extraction of CFGs from Open Systems
   4.1 Open Java bytecode systems
   4.2 The $\mathcal{O}$ Extraction Algorithm
   4.3 Correctness Proof

5 The ConFlEx Tool
   5.1 Implementation
   5.2 Experimental Results

6 Related work

7 Conclusion
   7.1 Discussion
   7.2 Future Work
CONTENTS

Bibliography

A Weak Simulation

B Correctness of $b^\gamma \circ BC2BTR$
Acknowledgements

It has been a long journey since I moved to a distant country, to start a new challenge. So many people helped me until now, and I hope I am fair in thanking them all.

Dilian Gurov: I cannot thank you enough for all the support you have given me. Not only you have been teaching me to do science, but have showed that it is made from people, and to the people. Siavash Soleimanifard: you are a great research partner, and also a great friend. Marieke Huisman and Afshin Amighi: thank you very much for the cooperation, hard work and patience. Attilio Picoco: you were a great research partner. It was a pleasure to work with you. Sérgio Campos and Alex Borges Vieira: thank you for the great collaboration, despite the distance. Roberto Guanciale: thank you very much for patiently reviewing the thesis. Mads Dam, Johan Hästad and Sonja Buchegger: thank you very much for the support. Andreas, Benjamin, Björn, Cenny, Douglas, Emma, Gourvan, Guillermo, Gunnar, Hamed, Jana, Karl, Lukas, Mateus, Muddassar, Musard, Fei, Ola, Oleksandr, Oliver, Sangxia, Shahram, Stephan, Tobias, Tobjörn: you have made the best research environment one can have. Thank you all for the support, and for the fun!

Dany, Raquel, David, Tiago, Felipe, Flávio, Alexandre, Ankur, Filipe, Sergej, Vesela, Susanne, and all the other friends in Sweden: thank you a lot for making me feel like home, even in such a cold weather. Paula: thank you very much for everything.

A warm "thank you" to all friends in Brazil (and some in other countries as well). I am lucky guy, and fortunately so many of you have visited me. Sorry if I could not spend all the time I wanted showing the beauties of Stockholm. Anyhow, from now on my visits to the Vasa museum are over!

To conclude, I dedicate this thesis to my family: mom (Patricinha), dad (Paulo), Elena and Lucas. I am happy man because I have you. Thank you very much for always being there.
Chapter 1

Introduction

Software systems are omnipresent in contemporary society. They are employed in virtually any area that affects people’s life. There are even areas where these systems are critical, and any failure would have a severe impact. Some examples are aircraft controller systems, banking transactions, or electronic voting. These sensitive applications have triggered an increasing demand for software quality and reliability.

In this context, formal methods, which are mathematically-based techniques, have gained growing acceptance as a means to ensure the trustworthiness of the critical software systems. A number of formal methods have been deployed in the last decades to formally verify software, such as various static analyses based on, for example, abstract interpretation, Hoare logic verification, and model checking. In contrast to testing methods, which uncover as many bugs as possible, but cannot guarantee their absence, formal verification techniques are exhaustive with respect to the property being verified.

Unfortunately, formal verification suffers from a combinatorial explosion of the state-space, as program size increases. The state-space explosion limits the range of properties that one can verify, and in most cases it even makes the verification intractable. A common approach to alleviate the problem is to generate an abstract model from the program, just preserving the information relevant to the properties of interest.

One common abstract program model are control flow graphs (CFG), where the control flow information is kept, and all program data is abstracted away (see e.g. [6, 36]). In a CFG, nodes represent the control points of the program, and the edges represent the move of control between control points. Numerous techniques have been proposed to extract automatically control flow graphs from program code (see e.g. [23, 8, 24]). Typically, however, these algorithms consider a very constrained subset of the program language at hand, or do not provide a correctness argument for the program abstraction. As a consequence, the produced models are not guaranteed to be sound, and the verification over such models may produce
wrong results.

One major goal of this work is to extract sound CFGs from Java bytecode (JBC), which is the executable language of the Java virtual machine (JVM). We focus on JBC because we want to extract CFGs even in the absence of the source code. For example, one may want to verify a system in which one of the components is provided by a third-party, but only as target code. In addition, we avoid any possible compiler-related issues, and we can analyze code written in other programming languages than Java that also compiles into JBC, such as Scala [31].

We take into account all the aspects of JBC which are relevant to control flow. First, the JBC is a stack-based language. It means that the operands for its instructions are stored in a stack, in contrast to the usual register-based approach. This imposes specific challenges to its static analysis. Moreover, despite being an executable language, the JBC contains several features of an object-oriented programming (OOP) language which makes it hard to predict the control flow. One feature is the subtype polymorphism, which allows different methods, known as virtual, to have the same signature. We call virtual method call (VMC) resolution the set of algorithms that estimates the possible receivers to a virtual invocation. Typically such algorithms enumerate the receivers by suitably analyzing a program’s class hierarchy, and its instructions.

Another feature that affects the control flow are exceptions, which are objects raised during the program execution to indicate an abnormal condition. They can be raised implicitly, by some internal failure (e.g. division by zero), or explicitly, by the `throw` instruction. Either the exception is caught, meaning that there was a specific (handler) code block to treat it, or the method execution terminates abruptly. In the latter case, the exception is propagated to the caller method, which is then in charge of handling the exception.

The control flow analysis considering exceptions is challenging for several reasons. First, the stack-based nature of the JVM makes it difficult to determine statically the type of the exceptions thrown explicitly. Thus, it is difficult to decide to which handler (if any) the control will be transferred. Second, the JVM can raise (implicit) run-time exceptions, such as `NullPointerException` and `IndexOutOfBoundsException`, by the abnormal execution of some of its instructions. To keep track of where such exceptions can be raised requires much care. Also, if a method does not handle an exception, its execution is aborted, and the exception handling is propagated to its caller method. The computation of control flow caused by exception propagation is not trivial because of the inter-dependencies between the program’s methods. Similar works have neglected the exceptional control flow because of the complexity it adds [43, 24].

The second major goal of this work is the analysis of open Java bytecode systems. We say that a software system is open if at least one of its components is missing. Typically, an open system is not executable until all of its components arrive, and it becomes closed. Still, one may want to analyze the available components. One framework that can analyze open systems is the compositional verification of control flow safety properties developed by Gurov et al. [15, 20, 19, 14], and implemented
as the CVPP tool-set [21] and its wrapper tool ProMoVer [38, 39, 37]. In this scenario, we want to produce CFGs from the available components that are sound over-approximations of the CFGs for the same components in any closed system that implements the original open system.

In this thesis we present two versions of a CFG extraction algorithm. The first one is for closed Java bytecode programs. The extraction algorithm considers all the typical intricacies of Java, such as virtual method call resolution, the differences between dynamic and static object types, and exception handling. The algorithm is defined indirectly, using the transformation into an intermediate representation of the JBC, named BIR. We have chosen the BIR because its stack-less representation facilitates the type analysis of exceptions raised explicitly by the user (with athrow instruction). Also, because its transformation inserts assertions along the code, which show the control points where an implicit (run-time) exception can be raised (e.g. NoSuchElementException). Next, we define a CFG extraction function from the BIR representation. Then we show the correctness of the composition of the BIR transformation with extraction function, i.e., we show that the CFGs it extracts structurally simulate the CFGs from a previous idealized extraction algorithm, which has been proven to simulate the JVM behaviorally. We reuse a previous result, which states that structural simulation entails behavioral simulation, to conclude that the indirect algorithm also simulates the JVM behavior.

We present a second version of the algorithm, which extracts CFGs for the available components of an open Java bytecode system. First, we extend the formal Java bytecode framework to model open programs. The missing components are represented by user-provided interfaces, that provide the relevant information about the component w.r.t analysis of the control flow. Next, we generalize the algorithm defined for closed system, to extract CFGs from the available components of an open system. It is also defined indirectly, and uses the BIR transformation. The inter-dependencies that involve missing components are resolved by using the information from the interfaces. The missing components will eventually be instantiated with concrete implementation, refining the open system until the point it becomes close, and can execute. Thus, we formally define a refinement relation between two open systems, which constrains the incoming components. Among other constraints, it checks if the instantiated components respect their interfaces. We conclude by showing that, whenever the refinement relation holds, it implies that the CFGs extracted from the original open system are sound over-approximations of the CFGs for the same components in the refined system. Thus, the verification results of temporal safety properties over such models are still valid for any closed system that implements the original open system.

We implement both versions of the extraction algorithm as the ConFLEx tool. It uses SawJa, a library for the static analysis of Java bytecode, for the BIR transformation and virtual method call resolution. We have tailored SawJa to address our needs. First, the BIR transformation has been improved to provide a more accurate type analysis of the exceptions raised explicitly. Originally, SawJa could not analyze open programs. We have extended it to support open Java
bytecode systems. This includes the implementation of a sound VMC resolution algorithm for modular set-ups, data structures to represent open Java bytecode programs, and the verification of the refinement relation.

Finally, we have implemented the CFG extraction algorithms from BIR in the ConFlEx tool. The extraction is dived into two stages: the first is the intra-procedural analysis, which extracts edges analyzing only the method’s instructions. The second is the inter-procedural analysis, which is a fixpoint computation of the exceptional control flow caused by exception propagation. The tool is able to cache previous analyses. This allows the exclusive extraction of newly-instantiated components, in contrast to an entire new extraction of the open system.

We have tested ConFlEx on several test-cases. The experimental results have shown that the algorithm for closed systems is much more precise than the algorithm for open systems, as expected. We have identified a correlation between the number of missing components, and the increase in the number of nodes and edges in a method’s CFG, due to the over-approximations the tool performs. Both algorithms have similar efficiency, and are linear w.r.t the number of instructions. Also, the fixpoint computation of control flow caused by exception propagation has proven to be lightweight in practice, and contributes to a negligible fraction of the total extraction times. Finally, the caching of previous analysis, which allows the exclusive extraction of the newly arrived component, improves significantly the extraction time of open systems when compared to an entire new analysis.

1.1 Contribution

The work presented in the thesis provides a comprehensive study for the generation of control flow graphs which are suitable for input to formal methods. To the best of our knowledge, this is the first work that presents a correctness argument for the extraction of control flow graphs from a real-world language. Also, we present a novel framework for the analysis of incomplete Java bytecode programs. We prove that our modular approach is sound, and also precise, in the sense that it only over-approximates when necessary. This makes the framework suitable for the analysis of real-world JBC programs.

The main contributions of this thesis are the following.

- An algorithm for the extraction of control flow graphs from closed Java bytecode programs considering precise exceptional flow. The algorithm has been proven to be sound with respect to the JVM behaviour. To the best of our knowledge, this is the first control flow graph extraction algorithm which both has been proven correct, and also has been fully implemented.

- A generalization of the algorithm above, for the extraction of control flow graphs from open Java bytecode programs. We introduce a formal JVM framework for open systems, and refinement rules, to preserve the soundness
of the algorithm. It has been proven to extract control flow graphs that simulate structurally any closed system that is implemented from the initial open system. To the best of our knowledge, this is the first extraction algorithm which produces sound control flow graphs for the available components of an open system. Also, it is the first one to produce control flow graphs which resolve all issues related to dynamic dispatching and exceptions for a real-world, object-oriented language, in a modular set-up.

- The Control Flow graph Extractor tool (ConFlEx), which implements the extraction algorithms for closed and open Java bytecode programs. To the best of our knowledge, the tool is the first one to implement control flow extraction algorithms which have a correctness argument. Moreover, it is the first to soundly extract control flow graphs for open systems. Thus, the tool is ideal for constructing models for formal software verification, especially for compositional verification of control flow safety properties.

The work presented in this thesis has generated the following scientific articles in conference proceedings.


The following articles are being prepared for submission to peer-reviewed publications:


The following technical reports have been produced along the work presented in the thesis.


As part of the current work, I have supervised one Master project, which resulted in the following thesis.


The following scientific articles have been published but are not part of the thesis.


My contributions. The work presented in this thesis, and all the artifacts it has generated, are result of scientific collaborations. The starting point was the idealized direct algorithm from Java Bytecode defined by Afshin Amighi, and its partial implementation in his extraction tool. I proposed together with Afshin the extraction of CFGs using the BIR transformation, and we sketched the strategy used to prove the behavioral simulation. Next I defined formally the extraction algorithm, and presented the structural simulation proof, with the assistance of Marieke Huisman and Dilian Gurov. I was the main author of all publications about the extraction algorithm for closed systems, but Dilian, Marieke and Afshin had equivalent contributions to mine. I re-wrote most of Afshin’s initial tool, and implemented the indirect algorithm for closed Java bytecode systems in the first version of ConFlEx.

I proposed and developed the generalization of the previous algorithm for open systems. During the conception, I had several discussions with Attilio Picoco, who contributed with ideas, and insights about practical matters. Also, Dilian constantly revised and criticized my ideas. I am the main author of the technical report about the analysis of open systems, but Attilio Picoco had an equivalent participation. I discussed jointly with Attilio the design choices for extending ConFlEx to support open systems. However, the majority of program code has been written by him, under my constant supervision.
1.2 Organization

The thesis is organized as follows. Chapter 2 summarizes results from previous works, which are necessary to the comprehension of our work. It also presents one technique that benefits from our work, and some motivating examples. Chapter 3 presents the extraction algorithm for closed Java bytecode systems, and presents its correctness argument. Chapter 4 presents a formal framework to represent open Java bytecode systems, generalizes the extraction algorithm for this set-up, and proves its correctness. Chapter 5 describes the implementation of the extraction algorithms for closed and open programs, as the ConFlEx tool. It presents experimental results, and compares both algorithms. Chapter 6 discusses the related work, and compares it to our approach. Finally, Chapter 7 summarizes our work and results, and presents possible directions for future work.
Chapter 2

Background

This chapter summarizes the previous work which serves as a base for this thesis. The first one is the formal definitions for the Java virtual machine, and bytecode, by Freund and Mitchell [13]. We present the main aspects which influence the control flow. The second work presents the definitions of the abstract program models we extract, as defined by Huisman et al. [19]. The third work is an idealized extraction algorithm from Java bytecode, defined by Amighi [1]. We use his algorithm as a reference to prove the correctness of ours. The forth work describes the BIR, an intermediate representation of the Java bytecode, present by Demange et al. [12]. Our algorithms uses the transformation from Java bytecode into BIR because of its support for exceptions. We conclude by briefly describing one compositional verification tool-set that benefits by a sound control flow extraction, and introduce some illustrative case studies.

2.1 Formal Java Virtual Machine Framework

In this section we present an overview of the formal Java virtual machine framework defined by Freund and Mitchell [13]. The work considers a significant fragment of the Java bytecode instructions set, which captures most of challenges on its static analysis. Specifically, virtual and interface method calls, and exceptions are featured. We summarize such definitions, focusing only in the relevant aspects to the control flow analysis.

A compiler that targets Java bytecode generates class files, one for each declared class, or interface. Each class declaration contains a symbolic name, type information, and the declaration of its method and fields. Let Class-Name and Interface-Name be the (countably) infinite sets of all class and interface names, respectively. Bytecode programs use method references, interface method references and field references to identify method, interface method and fields. These references are triples which describe the method (or interface method) in which it was declared, the method (or field) signature, and its type. They are generated from
the grammar in Figure 2.1.

\[
\begin{align*}
\text{Method-Ref} & ::= \{ | Class-Name, Label, Method-Type | \}_M \\
\text{Interface-Method-Ref} & ::= \{ | Interface-Name, Label, Method-Type | \}_I \\
\text{Field-Ref} & ::= \{ | Class-Name, Label, Field-Type | \}_F
\end{align*}
\]

Figure 2.1: Grammar generating references

In this work we consider a subset of the JVML\(_f\) instruction set described in \[13\]. Although it is significantly smaller, the subset contains one representative for each of the distinct cases to static analyze the control flow. For example, we have omitted the \textit{invokeinterface} instruction, since the control flow analysis for its case is analogous to \textit{invokevirtual}. The instructions \texttt{jsr q} and \texttt{ret r} for subroutine are not considered because they are deprecated since the JBC version 1.6 \[26\]. Figure 2.2 shows the bytecode instructions set considered in our project. We use the symbol \(x\) to denote a local method variable, and \(p\) to an instruction address.

\[
\begin{align*}
\text{Instruction} & ::= \text{nop} | \text{push} \; c | \text{pop} | \text{dup} | \text{add} | \text{div} \\
& | \text{if} \; p | \text{goto} \; p \\
& | \text{load} \; x | \text{store} \; x \\
& | \text{new} \; \text{Class-Name} \\
& | \text{athrow} \\
& | \text{getfield} \; \text{Field-Type} | \text{putfield} \; \text{Field-Type} \\
& | \text{invokespecial} \; \text{Method-Ref} \\
& | \text{invokevirtual} \; \text{Method-Ref} \\
& | \text{vreturn} | \text{return}
\end{align*}
\]

Figure 2.2: Subset of the JBC instructions

Java bytecode is a stack-based executable language. That is, most of the operands for its instructions are stored in the \textit{operand stack}. For example, the \texttt{if x} instruction branches to position \(x\) if the value on the top of the stack is zero. Also, the exception being raised by the \texttt{athrow} instruction, or the object whose method is being called by the \texttt{invokevirtual}, are also on top of the operand stack.

The JBC semantics, as in most semantics frameworks, models a program as an environment. Figure 2.3 shows the definition of an environment \(\Gamma\) which is the union of the partial mappings from classes and interfaces names, and method references, to their respective definitions. A class is defined by its parent class, the set of interfaces it implements, and its field. One interface contains the set of interfaces it inherits from, and the set of methods it provided. A method is defined by its array of instructions, and a list of exception handlers. An exception handler is 4-tuple \(\langle b, e, t, \sigma \rangle\), where \(b, e\) is the address range covered by the handler, \(t\) is
2.1. FORMAL JAVA VIRTUAL MACHINE FRAMEWORK

the address of the control point which handles the exception, and \( \sigma \in \text{Class-Name} \) is the exception type.

\[
\Gamma^I : \quad \text{Interface-Name} \rightarrow \langle \text{interfaces} : \text{set of Interface-Name}, \text{method} : \text{set of Interface-Method-Ref} \rangle \\
\Gamma^C : \quad \text{Class-Name} \rightarrow \langle \text{super} : \text{Class-Name}, \text{interfaces} : \text{set of Interface-Name}, \text{fields} : \text{set of Field-Ref} \rangle \\
\Gamma^M : \quad \text{Method-Ref} \rightarrow \langle \text{code} : \text{Instruction}^+, \text{handlers} : \text{Handler}^* \rangle \\
\Gamma = \Gamma^I \cup \Gamma^C \cup \Gamma^M
\]

Figure 2.3: Environment \( \Gamma \) of a Java program

The standard Java virtual machine contains a bytecode verifier (JBV), which performs several sanity checks on the code before the execution starts. It checks the correctness of the code format, if a method always terminate with a \texttt{return} or \texttt{throw} instruction, and if branches are to valid positions, among other analyses. The definition below states that a well-formed program is the one that passes successfully though all the verification tasks. In this work we assume that the input bytecode is always well-formed.

\textbf{Definition 1} (Well-Formed Java Program). A well-formed Java bytecode program is a closed program which passes the JVM bytecode verification. The exhaustive list of verification task is presented in \cite{41}.

Along an execution of the JVM, an active method is represented by an activation record. This is a 5-tuple which contains the method’s reference \( m \), the address \( p \) of the next instruction to be executed, a map \( f \) from the local variables to values, the method’s operand stack \( s \), and \( z \) is the information about the initialization of the object. The records are placed in the call stack, which stores in which sequence the methods are invoked. The top of the call stack contains the activation record of the current method being executed, or the record \( (e)_{\text{exc}} \), representing the case when an exception is raised. Figure 2.4 shows the syntax for the call stack.

It is important at this point to make a clear distinction between the operand stacks, and the call stack. An operand stack is defined for each method, and stores the values used by its instructions. A call stack is unique for a given JVM sequential program, and stores the records for the current active methods. In summary: a
JVM execution contains a single call stack, which by its turn may contain several operand stacks.

An execution state of the Java virtual machine is defined as a configuration $C = A; h$, where $A$ is the call stack, and $h$ represents a memory heap. The JVM behavior is the infinite-state transition system where the states are all the possible configurations, and the transition relation is defined by the operational semantics of the JBC instruction set, as presented in [13].

The Java bytecode is an executable language. Nevertheless, it contains some aspects of an object-oriented programming language. One is inheritance, which is the code reuse mechanism that allows one class to extend the definitions of another existing class. An environment has the inheritance definitions in $\Gamma^C$.interfaces and $\Gamma^I$.interfaces, which contains the interfaces a class or an interface will extend, and in $\Gamma^C$.super, which tells from what parent class a child class extends. The inheritance defines a type hierarchy between classes and interfaces. Every JBC program has a class hierarchy, being the class java.lang.Object the root.

The inheritance is transitive in JBC programs. That is, one class or interface inherits in cascade from its immediate classes and interfaces. The subtyping relation, defined for two class or interfaces $\tau_1$ and $\tau_2$ holds whenever $\tau_1$ inherits transitively from $\tau_2$. We use the notation $\Gamma \vdash \tau_2 < : \tau_2$ to denote the a subtyping holds for a given environment. Figure 2.5 shows the rules for the subtyping relation.

The subtyping plays a key role in the control flow analysis. First, because of the polymorphism, another OOP feature of bytecode. Polymorphism is possibility to have more than one implementation for the same method signature. In JBC, it is presented as subtype polymorphism. That is, it is possible for several classes in a sub-hierarchy to have the same method signature, but with a different implementation. We call those methods as virtual.

The invocation of virtual method is executed by the invokevirtual instruction, which operates over two parameters. One is the Method-Ref, which is hard-coded in the bytecode. The Method-Ref declaration contains the method signature, and the Class-Name, which we say to be the static type of the method. However, the second parameter is on the top of the operand It contains an object reference, and the dynamic type this object is what determines which of the polymorphic method implementations will be invoked. The exact dynamic type can only be determined in run-time. The only guarantee, provided by the JBV, is that the possible dynamic types are always sub-types of the static type. Virtual method call (VMC) resolution algorithms determines statically set of possible receivers for a given virtual invocation.

$\begin{align*}
A & ::= A' \mid \langle e \rangle_{exc} \cdot A' \\
A' & ::= \langle m, p, j, s, z \rangle \cdot A' \mid \epsilon
\end{align*}$

Figure 2.4: Syntax of the JVM call stack
2.2. PROGRAM MODELS

Exceptions are objects used to signal some abnormal condition during the program execution. In JBC, exceptions are objects whose class is a subtype of the java.lang.Throwable class. The exception classes are the ones present in the standard Java API, or user-defined. Also, an exception can either be raised explicitly by the user, or implicitly, by the erroneous execution of some instruction (e.g., division by zero). Explicit exceptions are raised with the athrow instruction. Its only operand is the reference to the exception to be thrown, which is on the top of the operand stack. Thus, static analyses have to perform some stack evaluation to determine the possible types of exceptions.

After the raise of an exception, the JVM verifies if there exists a suitable code block to handle it. This check searches for the first handler on the method’s handler table whose address range contains the address of the control point where the exception was raised, and its type is a sub-type of the exception. If a suitable handler is found, the control is transferred to the first instruction in that block; otherwise the current method is terminated abruptly, and the exception is propagated to the calling method, which now should handle the exception. This process continues until one of the methods in the stack of method invocations handles the exception, or the program terminates.

2.2 Program Models

Control flow graphs are an abstract model of a program. To define the structure and behavior of a CFG we follow Gurov et al. and use the general notion of
Definition 2 (Model, Initialized Model). A model is a (Kripke) structure \( \mathcal{M} = (S, L, \rightarrow, A, \lambda) \) where \( S \) is a set of states, \( L \) is a set of labels, \( \rightarrow \subseteq S \times L \times S \) a labeled transition relation, \( A \) a set of atomic propositions and \( \lambda : S \rightarrow P(A) \) a valuation assigning the set of atomic propositions that hold on each state \( s \in S \). An initialized model is a pair \((\mathcal{M}, \mathcal{E})\) with \( \mathcal{M} \) a model and \( \mathcal{E} \subseteq S \) a set of entry states.

Method graphs are the basic building blocks of control flow graphs. Let Method-Ref be the infinite set of all possible method signatures, and Excp-Name \( \subseteq \) Class-Name be the infinite set of all exceptions classes in Java. We define a method graph for sequential programs with procedures and exceptions as the instantiation of an initialized model, as follows.

Definition 3 (Method Graph). A method graph with exceptions for \( m \in \text{Method-Ref} \) over sets \( M \subseteq \text{Method-Ref} \), and \( E \subseteq \text{Excp-Name} \) is an initialized model \((\mathcal{M}_m, \mathcal{E}_m)\), where \( \mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m) \) with \( V_m \) the set of control nodes of \( m \), \( L_m = M \cup \{\varepsilon, \text{handle}\} \) the set of labels, \( A_m = \{m, r\} \cup E \), \( m \in \lambda_m(v) \) 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 node is tagged with the method signature it belongs to, and at most one exception. \( \mathcal{E}_m \subseteq V_M \) is a non-empty set of entry control point(s) of \( m \).

A method graph represents the control flow structure of a method. On it, nodes represent the control points of the method, and transitions represent the transfer of control between the control points. The set \( \mathcal{E}_m \) contains the node relative to the entry point of a method. Nodes tagged with the atomic proposition \( r \) represent return control points. A node can be either normal, having no exception as atomic proposition, or exceptional, having exactly one exception. The transitions are labeled either by a method signature (denoting a method call), by \( \text{handle} \) (to denote the handling of an exception), or by \( \varepsilon \) (to denote invisible actions).

Every control flow graph comes with an interface, which defines: the methods that are provided to, and required from the environment, the exceptions that a method may propagate, and the set of entry methods. The later is an empty set, for the methods which are not entry methods; if they are, then it is a unitary set with the method’s signature.

Definition 4 (Control Flow Graph Interface). A Control Flow Graph interface is a quadruple \( I = (I^+, I^-, E, M_e) \) where \( I^+, I^- \subseteq \text{Method-Ref} \) are finite sets of provided and required method signatures, \( E \subseteq \text{Excp-Name} \) is a finite set of exceptions, and \( M_e \subseteq \text{Method-Ref} \) is the set of entry methods (starting points of the program), respectively. If \( I^- \subseteq I^+ \), then \( I \) is closed.

Let \( \sqcup \) denote the standard disjoint union of two initialized models, and \( \cup \) denote the per-element union of two \( n \)-tuples of same type. We define a method’s control flow graph as the pair of its method graph and interface, and the control flow
2.2. PROGRAM MODELS

graph of a program is composed by the control flow graphs of all its methods. The composition of two control flow graphs is defined as follows.

**Definition 5 (Control Flow Graph Structure).** A Control Flow Graph \( G \) with interface \( I \), written \( G : I \) is inductively defined by:

- \((M_m, E_m) : (\{m\}, I^-, E, M_e)\) if \((M_m, E_m)\) is a method graph for \( m \) over \( I^-\), \( E \) and \( M_e \),
- \( G_1 \oplus G_2 : I_1 \cup I_2 \) if \( G_1 : I_1 \) and \( G_2 : I_2 \).

Figure 2.6 shows an example program. We present it as Java source code for simplicity. It has only single class, containing three methods. The column on the left represents the corresponding control points in the JBC program.

```java
public class Number {
    public static void main(String[] argv) {
        Number n = new Number();
        int myarg = Integer.parseInt(argv[1]);
        n.even(myarg);
    }

    public boolean odd(int n) {
        if (n < 0)
            throw new ArithmeticException();
        else if (n==0)
            return false;
        else
            return even(n-1);
    }

    public boolean even(int n) {
        try {
            if (n==0)
                return true;
            else
                return odd(n-1);
        } catch(ArithmeticException e) {
            n = n * (-1);
            }
        return odd(n-1);
    }
}
```

Figure 2.6: Example Java program with control points

Figure 2.7 shows the CFG for the example program in Figure 2.6. We use ◦ to denote a normal node, and ⋄ to stress a node tagged with an exception type. Notice that are several exceptional nodes in the CFG, and they do not have a corresponding control point on the source code. They represent that the control was taken by the JVM, to take care of the exception. The edges between an exceptional node to a normal one denote that there is a handler for the exception on that control point. The exceptional nodes tagged with the \( r \) atomic proposition denote an exception propagated by the method.

A CFG structure induces a behavior, which is the push-down automata used to model the JVM call stack. The Definition below extends the CFG behavior introduced in [19], to model the exceptional control flow.
Definition 6 (Control Flow Graph Behavior). Let $G = (M,E) : I$ be a closed Control Flow Graph with exceptions such that $M = (M,L,\to,A,\lambda)$. The behavior of $G$ is described by the initialized model $b(G) = (M_b,E_b)$, where $M_b = (S_b,L_b,\to_b ,A_b,\lambda_b)$ such that:

- $S_b \in V \times (V)^*$, i.e., states are pairs of control nodes and stacks of control nodes,
- $L_b = \{\tau\} \cup L_0^C \cup L_0^X$ where $L_0^C = \{m_1 \mid m_2 \mid l \in \{\text{call, ret, xret}, m_1, m_2 \in I^+\}$ (the set of call and return labels) and $L_0^X = \{lx \mid l \in \{\text{throw, catch}, x \in \text{Excp-Name}\}$ (the set of exceptional transition labels).
- $b = A$ and $\lambda_b((v,\sigma)) = \lambda(v)$
- $\to_b \subset S_b \times L_b \times S_b$ is the set of transitions in CFG$_m$ with the following rules:

\[
\begin{align*}
\text{[call]} & \quad \varepsilon \mid e \to_b (v_1, \sigma) \quad \text{[return]} & \quad (v_2, v_1, \sigma) \quad \text{[xreturn]} & \quad (v_2, v_1, \sigma) \\
\text{[transfer]} & \quad (v, \sigma) \quad \text{[throw]} & \quad (v, \sigma) \quad \text{[catch]} & \quad (v, \sigma)
\end{align*}
\]

Each pair is made of a control node and the current configuration of the node stack, to keep the order of the transitions and the states visited. The silent transitions, labeled with $\varepsilon$, induce a control shift from one normal state to another, as defined by the [transfer] rule. The [throw] rule refers to an $\varepsilon$ transition as well, but the next state node to visit is marked as exceptional. Both [catch] and [xreturn] rules are induced by a handle-transition. The former implies that a specific handler code block is going to catch explicitly the raised exception. The latter means that the control flow is going to move from the called method to the calling one because
Figure 2.7: Example program’s control flow graph

of a raised exception, not caught in the returning method but potentially handled in the invoking method. The couple \([\text{call]}-\text{[return]}\) refers to the case when a method is invoked by another method and the called returns to the calling one. In both \([\text{return}]\) and \([\text{xreturn}]\), the destination state in the returning method is marked as a return node.

Now we show how the induced CFG behavior models the JVM behavior. We define the abstraction function \(\theta\), which that maps a JVM configurations to CFG behavioral configuration, as follows.

**Definition 7** (Abstraction Function for VM States). Let \(\text{Conf}\) be the set of JVM execution configurations and \(S_b\) the set of states in \(b(G)\). Then \(\theta : \text{Conf} \rightarrow S_b\) is defined inductively as follows:

\[
\theta(c) = \begin{cases} 
\langle \sigma^m_p, \epsilon \rangle & \text{if } c = ((m, p, f, s, z), \epsilon; h) \\
\langle \sigma^m_p, \theta(A; h) \rangle & \text{if } c = ((m, p, f, s, z), A; h) \\
\langle \bullet^x_m, \theta(A; h) \rangle & \text{if } c = ((x)^\text{exc}, (m, p, f, s, z), A; h) \\
\langle \bullet^x_m, \epsilon \rangle & \text{if } c = ((x)^\text{exc}, \epsilon; h) 
\end{cases}
\]

Function \(\theta\) is defined recursively, and applies to all activation records on the call stack. The symbol \(\bullet\) denotes the special abort control point, which is reached only when the call stack is empty, caused by an uncaught exception.

### 2.3 Direct Extraction of CFGs from Bytecode

This section briefly describes the idealized direct extraction algorithm presented by Amighi et al. [1]. The CFGs extracted from the algorithm have been proven to simulate the JVM behavior. We use this correctness result to establish the correctness
of our algorithms. We prove the structural simulation of our algorithms to the idealized one. Then, we conclude the correctness of our algorithms by reusing a previous result which states that structural simulation implies behavioral simulation [15].

The direct algorithm groups the JBC instructions into disjoint sets (Figure 2.8). In JBC, the instruction `athrow` (explicit exception throw) does not have an argument; instead the exception is determined at run-time by the object reference on top of the stack. However, the algorithm replaces `athrow` by the idealized instruction `throw X`, which considers the static type `X` of the exception as the operand. Some static analysis technique can determine the static type `X`. In fact, that is the information that the BIR transformation computes, as shown in Section 2.4.

<table>
<thead>
<tr>
<th>Subset</th>
<th>Description</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetInst</td>
<td>Normal return instructions</td>
<td><code>return</code></td>
</tr>
<tr>
<td>CmpInst</td>
<td>Computational instructions</td>
<td><code>nop, push v, pop</code></td>
</tr>
<tr>
<td>CndInst</td>
<td>Conditional instructions</td>
<td><code>ifeq q</code></td>
</tr>
<tr>
<td>JmpInst</td>
<td>Jump instructions</td>
<td><code>goto q</code></td>
</tr>
<tr>
<td>XmpInst</td>
<td>Potentially can raise exceptions</td>
<td><code>div, getfield f</code></td>
</tr>
<tr>
<td>InvInst</td>
<td>Method invocations</td>
<td><code>invokevirtual (o,m)</code></td>
</tr>
<tr>
<td>ThrInst</td>
<td>Explicit exception throw</td>
<td><code>throw X</code></td>
</tr>
</tbody>
</table>

Figure 2.8: Grouping of bytecode instructions

The nodes of a method’s CFG are uniquely identified by the method signature, a program counter, and set of atomic propositions which hold on it. Based on Definition 3 to construct the nodes we have to specify `V_m`, `A_m`, `λ_m`, and `E_m`. For a node `v ∈ V_m` indicating control point `ℓ ∈ Addr`, of method `m`, we define `v = (m, ℓ)`.

The labeling function `λ_m` specifies `A_m` for a given `v ∈ V_m`. If `m[ℓ] ∈ RetInst` then the node is tagged with `r`. If the node is an exceptional node `v` then it is marked with the exception type `x ∈ E`. The method signature is the default tag for all the method’s control nodes. If `ℓ = 0` then the node will be a member of `E_m`.

Two nodes are equal if they specify the same control address of the same method, with equal atomic proposition sets. We use the following notations: `•_x` stresses a control node tagged with an exception `x`, and `◦_n` denotes a normal control node.

The subset of nodes for method `m` is denoted as `V_m ⊆ V` and `(v_1 l →_m v_2)` is an edge of the method `m`’s CFG labeled with `l`.

The CFG extraction rules for method `m` in environment `Γ` use the implementation of the method: `Γ[m] = (P, H)`. For each instruction in `P` (body of `Γ[m]`), the rules build a set of labeled edges connecting control nodes.

**Definition 8 (Method Control Flow Graph Extraction).** Let `V_m ⊆ V` be the set of nodes and `L_m = M ∪ {ε, handle}`, `M ⊂ Meth`. Let `Π` be a set of environments. Then the control flow graph of method `m` is extracted by `mG : Π × Meth → P(V_m × L_m × V_m)`, defined in Figure 2.9 (where `succ` denotes the next instruction address function).
For simple computational instructions, a direct edge to the next control address is established. For jump instructions, an edge to the jump address \( X \) is the set of possible exceptions, identified by the transformation algorithm.

For instructions in \( \text{XMPINST} \) edges for all possible flows are added: successful execution and exceptional execution, with edges for successful and failed exception handling, as defined by function \( H_p^{\times} \). This function constructs the outgoing edges of the exceptional nodes by searching the exception handling table for a suitable handler of exception type \( x \) at position \( p \). If there is a handler, it returns an edge from an exceptional node to a normal node (handler). Otherwise it produces an edge to an exceptional return node to propagate the exception to the caller. If the method is an entry method \( (m \in M_e) \) then there is no caller to propagate the uncaught exception and the exceptional return node will be tagged with special address to denote the abnormal termination. Function \( h \) seeks the proper handler in the exception handling table; it returns 0 if there is no entry for the exception at the specified control point. The function \( X : \text{XMPINST} \rightarrow \mathcal{P}(\text{EXCP}) \) determines possible exceptions of a given instruction. The \( \text{throw} \) instruction is handled similarly, where \( X \) is the set of possible exceptions, identified by the transformation algorithm.

To extract edges for method invocations, function \( \text{RecT}(i) \) yields the set of possible method signatures of a method call in environment \( \Gamma \). Following defines the function formally:

\[
\text{RecT}(i) = \begin{cases} 
\{ n_{\text{staticT}(o)} \} & \text{if } i \in \{ \text{invokespecial} (o,n) \}, \text{invokestatic} (o,n) \\
\{ n_r \mid \tau \in \text{resT}_p^{\alpha}(o,n) \} & \text{if } i = \text{invokevirtual} (o,n)
\end{cases}
\]
Chapter 2. Background

The receiver object for `invokevirtual` is determined by late binding. The virtual method call resolution function $\text{res}_\alpha^\Gamma$ is employed, where $\alpha$ is a standard static analysis technique to resolve the call. We use $n_T$ to indicate method $n$ from class $T$. For example, Rapid Type Analysis (RTA) \cite{4} returns the set of subtypes of the callee’s static type which are instantiated in the program (created by a `new` instruction). So for $\alpha = \text{RTA}$, the result of the resolution for object $o$ and method $n$ in environment $\Gamma$ will be:

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

where $I_{C_\Gamma}$ is the set of instantiated classes in environment $\Gamma$, $\text{static}T(o)$ gives the static type of object $o$ and $\text{lookup}(n, \tau)$ corresponds to the signature of $n$, i.e., $\tau$ is a subtype of $o$’s static type and method $n$ is defined in class $\tau$.

Given the set of possible receivers, calls are generated for each possible receiver. For each call, if the method’s execution terminates normally, control will be given back to the next instruction of the caller. If the method terminates with an uncaught exception, the caller has to handle this propagated exception. The CFG extraction rules for method invocations produce edges for both $g_N = \text{NullPointerException}$ (possibly null receiver object exception) and for all propagated exceptions.

$$\mathcal{H}_{\text{N}}^\rho$$ is the set of edges to handle $g_N$, and $N_{\text{ip}}^\rho$ defines the set of edges to handle all uncaught exceptions from all possible callees. Similar to generating outgoing edges for exceptional control points, $\mathcal{H}_{\text{ip}}^\rho$ generates edges for successful/failed handlers for all exceptional nodes in $\text{CFG}_n$, i.e., the CFG of method $n \in \text{res}_\alpha^\Gamma(o,n)$.

The last rule in Figure 2.9 decomposes a sequence of instructions into individual instructions. For each individual instruction, a set of edges is computed. Then, the CFG of a method is the union of all edges produces from its instructions. The CFG of a Java class $c \in \text{Class}$ is defined as the disjoint union of the CFGs of the methods in $c$. The CFG of a Java bytecode program $\text{Prg}$, denoted $G_{\text{jbc}} : \Pi \rightarrow \mathcal{P}(V \times L_m \times V)$, is the disjoint union of all CFGs of the classes in $\text{Prg}$.

### 2.4 The BIR Language

The BIR language is an intermediate representation of Java bytecode. The main difference with JBC is that BIR instructions are stack-less. That is, instructions operate over local variables, and not over values stored on the operand stack, as in bytecode. The transformation into a stack-less representation simplifies the static analysis of the control flow, specially the exceptional flow. We give a brief overview of BIR; for a full account we refer to \cite{12}.

Figure 2.10 summarizes the BIR syntax. Its instructions operate over expression trees, i.e., arithmetic expressions composed of constants, operations, variables, and fields of other expressions ($\text{expr.f}$). BIR does not have operations over strings and booleans; these are transformed into methods calls by the BC2BIR transformation.
2.4. THE BIR LANGUAGE

expr ::= c | null
| expr ⊕ expr
| tvar | lvar
| expr.f

Assignment ::= target := expr

Return ::= return expr | return

MethodCall ::= expr ns(expr,...)

lvar ::= x | x1 | ... NewObject ::= target := new C(expr,...)

this

Assertion ::=nonnull expr | notzero expr

tvar ::= t | t1 | ... instr ::= nop | if expr pc | goto pc

target ::= lvar
| expr.f

MethodCall | NewObject
| Assignment | Return

| throw expr | mayinit C

BIR has two types of variables: var and tvar. The first are identifiers also present in the original bytecode; the latter are new variables introduced by the transformation. Both variables and object fields can be an assignment’s target.

As a convention, we use brackets to distinguish BIR instructions. Many of the BIR instructions have an equivalent JBC counterpart, e.g., [nop], [goto] and [if]. A [return expr] ends the execution of a method with a return value, while [return] ends a void method. The [throw] instruction explicitly transfers control flow to the exception handling mechanism, similarly to the athrow instruction in JBC. Method call instructions are represented by their method signature. For non-void methods, the instruction assigns the result value to a variable.

In contrast to JBC, object allocation and initialization happen in a single step, during the execution of the [new] instruction. However, Java also has class initialization, i.e., the one-time initialization of a class’s static fields. BIR has the special instruction [mayinit] to indicate that at that point a class may be initialized for the first time. Otherwise, it behaves exactly as [nop].

BIR’s support of implicit exceptions follows the approach proposed for the Jalapeño compiler [7]. It inserts special assertions before the instructions that can potentially raise an exception, as defined by the JVM. Figure 2.11 shows all implicit exceptions that are currently supported by the BC2BIR transformation [5], and the associated assertion. For example, the transformation inserts a [nonnull] assertion before any instruction that might raise a NullPointerException, such as an access to a reference. If the assertion holds, it behaves as a [nop], and control flow passes to the next instruction. If the assertion fails, control flow is passed to the exception handling mechanism. In the transformation from BIR to CFG, we use a function χ to obtain the exception associated with an assertion. Notice that our
### Assertion | Exception
--- | ---
[nonnull] | NullPointerException
[checkbound] | IndexOutOfBoundsException
[notneg] | NegativeArraySizeException
[notzero] | ArithmeticException
[checkcast] | ClassCastException
[checkstore] | ArrayStoreException

Figure 2.11: Implicit exceptions supported by BIR, and associated assertions

translation from BIR to CFG can easily be adapted for other implicit exceptions, provided appropriate assertions are generated for them.

A BIR program is organized in exactly the same way as a Java bytecode program. A program is a set of classes, ordered by a class hierarchy. Every class consists of a name, methods and fields. A method’s code is stored in an instruction array, with indexing starting with 0 for the entry control point. However, in contrast to JBC, in BIR the indexes in the instruction array are sequential.

Next we give a short overview of the BC2BIR transformation. It translates a complete JBC program into BIR by symbolically executing the bytecode using an abstract stack. This stack is used to reconstruct expression trees and to connect instructions to its operands. As we are only interested in the set of BIR instruction that can be produced, we do not discuss all details of this transformation. For the complete algorithm, we refer to [12].

The symbolic execution of the individual instructions is defined by a function \( BC2BIR_{\text{instr}} \) that, given a program counter, a JBC instruction and an abstract stack, outputs a set of BIR instructions and a modified abstract stack. In case there is no match for a pair of bytecode instruction and stack, the function returns the \( \text{Fail} \) element, and the \( BC2BIR \) algorithm aborts. The function \( BC2BIR_{\text{instr}} \) is defined as follows.

**Definition 9 (BIR Transformation Function).** Let \( \text{AbsStack} \in \text{expr}^* \). The rules defining the instruction-wise transformation \( BC2BIR_{\text{instr}} : \mathbb{N} \times \text{instr}_{JBC} \times \text{AbsStack} \rightarrow ((\text{instr}_{BIR})^* \times \text{AbsStack}) \cup \{ \text{Fail} \} \) from Java bytecode into BIR are given in Figure 2.12.

JBC instructions if, goto, return and vreturn are transformed into corresponding BIR instructions. The new is distinct from \( \text{[new C()]} \) in BIR, and produces a [mayinit]. The getfield \( f \) instruction reads a field from the object reference at the top of the stack. This might raise a NullPointerException, therefore the transformation inserts a [nonnull] assertion.

The store \( x \) instruction produces one or two assignments, depending on the state of the abstract stack. Instruction putfield \( f \) outputs a set of BIR instructions: [nonnull \( e \)] guards if the \( e \) is a valid reference; then the auxiliary function
2.5. COMPOSITIONAL VERIFICATION

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>∅</td>
<td>nop</td>
<td>[nop]</td>
<td>div</td>
<td>[notzero e]</td>
</tr>
<tr>
<td>push c</td>
<td>∅</td>
<td>if p</td>
<td>[if e pc']</td>
<td>athrow</td>
<td>[throw e]</td>
</tr>
<tr>
<td>dup</td>
<td>∅</td>
<td>goto p</td>
<td>[goto pc']</td>
<td>new C</td>
<td>[mayinit C]</td>
</tr>
<tr>
<td>load x</td>
<td>∅</td>
<td>return</td>
<td>[return]</td>
<td>getfield f</td>
<td>[notnull e]</td>
</tr>
<tr>
<td>add</td>
<td>∅</td>
<td>vreturn</td>
<td>[return e]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input | Output
----|-------
store x | [x:=e] or [t₀ᵖᶜ:=x;x:=e]
putfield f | [nonnull e;FSave(pc,f,as);e.f:=e']
invokevirtual ns | [nonnull e;HSave(pc,as);t₀ᵖᶜ:=e.ns(e₁'...eₙ')]
invokespecial ns | [nonnull e;HSave(pc,as);t₀ᵖᶜ:=e.ns(e₁'...eₙ') or [HSave(pc,as);t₀:=new C(e₁'...eₙ')]]

Figure 2.12: Rules for BC2BIR

FSave introduces a set of assignment instructions to temporary variables; followed by the assignment to the field (e.f). Similarly, instruction invokevirtual generates a [nonnull] assertion, followed by a set of assignments to temporary variables – represented as the auxiliary function HSave – and the call instruction itself. The transformation of invokespecial can produce two different sequences of BIR instructions. The first case is the same as for invokevirtual. In the second, there are assignments to temporary variables (HSave), followed by the instruction [new C] which denotes a call to the constructor.

Figure 2.13 shows the JBC and BIR versions of method odd() (from Figure 2.6). The different colors show the collapsing of instruction by the transformation: the underlined instructions are the ones that produce BIR instructions. The BIR method has a local variable (x) and a newly introduced variable (t₀). Notice that the argument for the method invocation and the operand to the [if] instruction are reconstructed expression trees. The [nonnull] instruction asserts that NullPointerException can potentially be raised on that program point.

2.5 Compositional Verification of Control Flow Properties

In this section we motivate our work by presenting a formal verification technique that benefits from the algorithms and tools presented in the thesis. We describe the technique for compositional verification of control flow temporal safety properties introduced by Gurov et al. [15, 20]. The technique decouples the global correctness of a system from its implementation; the correctness is relativized over its components’ local properties. A global property is checked over a maximal control flow graph, constructed from the software components’ properties, and interfaces. A maximal control flow graph for a given property is the CFG that simulates any CFG that satisfies the property. Then, the verification of the local properties over
the implementation of each component guarantees the global correctness.

The decoupling of the verification of a global property from the program’s actual implementation allows the efficient verification of systems where the implementation evolves, or that some parts are not available. Let’s assume we want to verify an ATM system that depends on a third-party component to execute, yet unavailable. This component is present in the users’ smart-cards, and becomes available when a user inserts the card in the ATM. By decoupling, one can verify in advance a system’s global property, and the local properties of the available components. The only pending task is the verification of the component in the smart-card, which is lightweight, and it can be delayed until the user arrives to the ATM. If the verification fails, then the ATM denies any transaction.

Now we present the compositional verification technique. Let $\psi$ be the specification of a software component, and $I$ its interface. A maximal control flow graph for the pair is denoted as $Max(\psi, I)$. Also, let $G$ be the control flow graph from a given component, $k$ be the number of components for an arbitrary system, and $\phi$ be a global property. The compositional verification principle is presented as the following proof rule, with $k + 1$ premises.

$$
\begin{align*}
G_1 \models \psi_1 & \quad \ldots \quad G_k \models \psi_k & \quad \bigcup_{i=1,\ldots,k} Max(\psi_i, I_i) \models \phi \\
\bigcup_{i=1,\ldots,k} G_i \models \phi
\end{align*}
$$
2.5. COMPOSITIONAL VERIFICATION

The CFG of a closed program, as defined in Section 2.2, is the disjoint union of the CFGs from its components. Thus, the principle states that the program’s control flow graph satisfies the global specification if the latter is satisfied by the system’s composed maximal control flow graph, and if each component’s CFG satisfies its local specification.

The technique has been implemented as CVPP [14][21], a tool-set for the compositional verification of temporal control flow safety properties for Java bytecode programs. CVPP is wrapped by ProMoVer [38][39][37], a tool which encapsulate the verification steps, and provide a push-button interface for the user. In the context of ProMoVer, individual program methods define the granularity for components. The properties to be verified are specified in Linear Temporal Logic (LTL) [33], a temporal logic that reasons about sequences of events. ProMoVer can verify safety properties about sequences of method invocations, and exceptions.

In ProMoVer, the compositional verification principle gives rise to two parallel tasks: (i) the verification of local properties over CFGs from methods, and (ii) the verification of the global property over the maximal control flow graph. The local properties of software components are structural, i.e., reason about the CFG structure (Def. 5). The global property is behavioral, i.e., reasons about the CFG behavior (Def. 6). The two independent tasks are executed as follows.

(i) Check $G_i \models \psi_i$ for $i = 1, \ldots, k$: (a) extract CFGs $G_i$ for each component, and (b) model check $G_i$ against $\psi_i$. CFGs are represented as Kripke structures, and verified by means of standard finite-state model checking.

(ii) Check $\bigcup_{i=1,\ldots,k} \text{Max}(\psi_i, I_i) \models \phi$: (a) Construct maximal flow graphs $\text{Max}(\psi_i, I_i)$ for all method specifications $\psi_i$ and interfaces $I_i$, (b) compose the graphs, resulting in the maximal control flow graph $G_{\text{Max}}$, and (c) model check $G_{\text{Max}}$ against the global property $\phi$. For (c), the behavior of $G_{\text{Max}}$ is represented as a push-down system (PDS), and a standard PDS model checker is used.

ProMoVer is tailored for the verification of open systems. For example, assume that one wants to verify an open Java bytecode system with $k$ components. However, the component $i$ ($1 \leq i \leq k$) is missing. Still the global correctness of the system can be verified by the task (ii). This is the case of the ATM scenario, described above. The local correctness of the available components can be verified immediately. The only pending verification task would be the local verification of $G_i \models \psi_i$, which is postponed until the arrival of component $i$.

Example of compositional verification. Let’s suppose we have the Java program from Figure 2.6 but the implementation of the main method is not provided. We want to check if the program can abort the execution because an ArithmeticException is not caught. The following LTL formula expresses this property, essentially stating that the control never reaches a return control point of method main tagged with the exception type.

$$\phi = \Box(\neg(main \land \text{ArithmeticException}) \land r)$$
We define the local property $\psi$ of \texttt{main} informally as '\texttt{main} may only call \texttt{even}'. Next, we construct the composed maximal CFG for the system from $\psi$, and also from the local properties of the other methods (for simplicity, not mentioned here). The verification shows that the global property $\phi$ holds for the maximal CFG. Once the implementation of \texttt{main} is provided, we simply extract its CFG, and check it against the local property $\psi$. If it holds, then the correctness of the program is proved w.r.t to $\phi$. 
Chapter 3

Extraction of CFGs from Closed Systems

This chapter presents the two-phase transformation from Java bytecode into control flow graphs, using the BC2BIR transformation, presented in Section 2.4. First we define the extraction function of CFGs from BIR, which we call $b\mathcal{G}$. Next, we outline the correctness proof of the indirect algorithm: CFGs extracted with $BC2BIR \circ b\mathcal{G}$ structurally simulate CFGs extracted with the $m\mathcal{G}$ algorithm, presented in Section 2.3. We illustrate it by presenting one interesting case, and refer to the complete proof in Appendix B. We conclude by presenting the proof strategy, which shows that CFGs extracted indirectly simulate behaviorally the JVM.

3.1 Transformation from BIR into Control Flow Graphs

The extraction algorithm that generates a CFG from BIR iterates over the instructions of a method. It uses the transformation function $b\mathcal{G}$, that takes as input a program counter, an instruction for a BIR method, and its exception table. Each iteration outputs a set of edges.

To define $b\mathcal{G}$, we introduce some auxiliary functions, which are similar to the ones introduced for the direct extraction (in Section 2.3). As a convention, we use bars (e.g., $\bar{N}$) to differentiate the similar functions from the direct, and indirect algorithms.

First, let $\mathcal{E}_t$ be the set of all exception tables. $\bar{H} \in \mathcal{E}_t$ is the exception table for a given method, containing the same entries as the JBC table, but with control points relating to BIR instructions. The function $h_{\bar{H}}(pc, x)$ searches for the first handler for the exception of type (or a subtype of) $x$ at position $pc$. given a virtual method call resolution algorithm $\alpha$. The function $\bar{H}_{pc}$ returns an edge after querying $h_{\bar{H}}$: if there was an exception handler, it returns an edge to a normal control node; otherwise, it returns an edge to an exceptional return node.
The extraction is parametrized by a virtual method call resolution algorithm \( \alpha \). The function \( \text{res}^\alpha(\text{ns}) \) uses \( \alpha \) to return a safe over-approximation of the possible receivers to a virtual invocation of a method with signature \( \text{ns} \), or the single receiver if the signature is from a non-virtual method (e.g., to a static method).

We divide the definition of \( bG \) into two parts. The \textit{intra-procedural} analysis extracts for every method an initial CFG, based solely on its instruction array, and its exception table. Based on these CFGs, the \textit{inter-procedural} analysis computes the functions \( \tilde{N}_n^\text{pc} \), which return exceptional edges for exceptions propagated by calls to method \( n \). The functions for inter-dependent methods are thus mutually recursive, and are computed in a fixed-point manner.

**Definition 10** (Control Flow Graph Extraction). The control flow graph extraction function \( bG : (\text{Instr} \times \mathbb{N}) \times \mathcal{E} \rightarrow \mathcal{P}(V \times L_m \times V) \) is defined by the rules in Figure 3.1. Given method \( m \), with \( \text{ArInstr}_m \) as its instruction array, the control flow graph for \( m \) is defined as \( bG(m) = \bigcup_{i \in \text{ArInstr}_m} bG(i_{pc}, \tilde{H}_m) \), where \( i_{pc} \) denotes the instruction with array index \( pc \). Given a closed BIR program \( \Gamma_B \), its control flow graph is \( bG(\Gamma_B) = \bigcup_{m \in \Gamma_B} bG(m) \).

Figure 3.1: Extraction rules for control flow graphs from BIR

First, we describe the rules applied by the intra-procedural analysis. Assignments, \( \text{[nop]} \) and \( \text{[mayinit]} \) add a single edge to the next normal control node. The conditional jump \( \text{[if} \ expr \ pc' \text{]} \) produces a branch in the CFG: control can go either to the next control point, or to the branch point \( pc' \). The unconditional jump \( \text{go to} \ pc' \) adds a single edge to control point \( pc' \). The \( \text{[return]} \) and \( \text{[return} expr \text{]} \) instructions generate an internal edge to a return node, i.e., a node with the
atomic proposition \( r \). Notice that, although both nodes are tagged with the same \( pc \), they are different, because their sets of atomic propositions are different.

The \([\text{throw } X]\) instruction, similarly to virtual method call resolution, depends on a static analysis to find out the possible exceptions that can be thrown. The BIR transformation only provides the static type \( X \) of the thrown exception. Let \( X \) also denote the set containing the static type, and all its subtypes. The transformation produces an exceptional edge for each element \( x \) of \( X \), followed by the appropriate edge derived from the exception table.

The rule for assertion instructions produces a normal edge, for the case that the implicit exception is not raised, and an edge to the exceptional node tagged with the exception type (as defined in Figure 2.11), together with the appropriate edge derived from the exception table.

The extraction rule for a constructor call (\([\text{new} \ C]\)) produces a single normal edge, since there is only one possible receiver for the call. In addition, we also produce an exceptional edge, because of a possible \text{NullPointerException}.

The extraction rule for method calls is similar to that of the direct extraction. Again, we assume that an appropriate virtual method call resolution algorithm is used, we add a normal edge for each possible receiver returned from \( res^\alpha \).

Next, we describe the inter-procedural analysis. In all program points where there is a method invocation, the function \( \bar{N}_n^{pc} \) adds exceptional edges, relative to exceptions propagated by called methods. It analyzes if the CFG of an invoked method \( n \) contains an exceptional return node. If it does, then function \( \bar{H}_x^{pc} \) verifies whether the exception of type \( x \) is caught in position \( pc \). If so, it adds an edge to the handler. Otherwise it adds an edge to an exceptional return node. In the later case, the propagation of the exception continues until it is caught by some caller method, or there are no more methods to handle it. This is similar to the process described by Jo and Chang [24], who also present a fixed-point algorithm to compute the propagation edges. It checks the pre-computed call-graph which are the callers to a method propagating a given exception, and at which control points. If there is a suitable handler for that exception, it adds the respective handling edges, and the process stop. Otherwise, the computation proceeds.

### 3.2 Correctness of CFG Extraction

This section discusses the correctness proof of the CFG extraction algorithm. We prove correctness indirectly, using as reference the idealized direct extraction algorithm \( mG \) defined in Section 2.3, and proved correct in [1]. \( mG \) is based directly on the semantics of Java bytecode, but assumes an oracle to predict the exceptions that can be thrown by each instruction.

We exploit the idealized algorithm by proving that given a JBC program, the CFG produced by our extraction algorithm \( (bG \circ BC2BIR) \) structurally simulates the CFG produced by the direct extraction algorithm \( mG \). We then reuse an existing result from Gurov et al. [15] Th. 36 that structural simulation implies behavioral
simulation. By transitivity of simulation we conclude that the behavior induced by the CFG extracted by $bG \circ BC2BIR$ simulates the JVM behavior. Figure 3.2 summarizes our approach.

Figure 3.2: Schema for CFG extraction and correctness proof

The proof of structural simulation is too large to be presented completely in this section. Instead, we sketch the overall proof, and discuss one case (for the `athrow` instruction) in full detail. For the remaining detailed cases, the reader is referred to the Appendix. Before discussing the proof sketch, we first introduce some terminology and relevant observations.

**Preliminaries for the Correctness Proof** The BC2BIR transformation may collapse several bytecode instructions into a single BIR instruction. Therefore, we divide bytecode instructions as **producer** instructions, i.e., those that produce at least one BIR instruction in function $BC2BIR_{\text{instr}}$, and **auxiliary** ones, i.e., those that produce none. This division can be deduced from Figure 2.12 (on page 23). For example, `store` and `invokevirtual` are producer instructions, while `add` and `push` are auxiliary.

We partition the bytecode instruction array into **bytecode segments**. These are subsequences delimited by producer instructions. Thus each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java bytecode Verifier. All methods in such a program must terminate with `return`, or `athrow`, which are producer instructions. Therefore, there can not be a set of contiguous instructions that is not delimited by a producer instruction.

A **BIR segment** is the result of applying BC2BIR on a bytecode segment. Thus there exists a one-to-one, order-preserving mapping between bytecode segments and BIR segments, and we can associate each JBC or BIR instruction to the unique index of its corresponding bytecode segment.

Figure 2.13 (on page 24) illustrates the partitioning of instructions into segments. Method `odd` has four bytecode (and BIR) segments, as indicated by the coloring. Producer instructions are underlined.

In the definition of the direct extraction algorithm in Figure 2.9 (on page 19), one can observe that all auxiliary instructions give rise to an internal transfer edge.
3.2. CORRECTNESS OF CFG EXTRACTION

only. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path of internal transfer edges with the same size as the number of auxiliary instructions, followed by the edges generated for the producer instruction.

**Proof Sketch** Based on observations above, our main theorem states that the method graph extracted using the indirect algorithm weakly simulates (cf. [28]) the method graph using the direct algorithm. In the proof, we do not consider the abstract stacks, since only the instructions are relevant to produce the edges.

**Theorem 1** (Structural Simulation of Method Graphs). Let $\Gamma$ be a well-formed Java bytecode program, and let $\Gamma[m]$ be the implementation of method $m$. Then $(b\mathcal{G} \circ \text{BC2BIR})(\Gamma[m])$ weakly simulates $m\mathcal{G}(\Gamma[m])$.

**Proof.** (Sketch) Let $p$ range over indices in the bytecode instructions array, $pc$ over indices in the BIR instructions array, $p,m$ over control nodes in $m\mathcal{G}(\Gamma[m])$, and $pc,m,x,y$ over control nodes in $(b\mathcal{G} \circ \text{BC2BIR})(\Gamma[m])$. The control nodes are valuated with two optional atomic propositions: $x$, which is an exception type, and $y$, which is the atomic proposition $r$ denoting a return point. Further, let $\text{seg}_{JBC}(m,p)$ and $\text{seg}_{BIR}(m,pc)$ be two auxiliary functions that return the segment number that a bytecode, or a BIR instruction belongs to, respectively, and let function $\text{min}(s,x,y)$ return the least index $pc$ in the BIR segment $s$ resulting in a node valuated with $x$ and $y$.

We define a binary relation $R$ as follows:

$$R \overset{\text{def}}{=} \{(p,m,x,y), (p,m,x,y) \mid \text{seg}_{JBC}(m,p) = \text{seg}_{BIR}(m,pc) \land pc = \text{min}(\text{seg}_{BIR}(m,pc), x, y)\}$$

and show the relation to be a weak simulation in the standard fashion: for every pair of nodes in $R$, we match every strong transition from the first node by a corresponding weak transition from the second node, so that the target nodes are again related by $R$. It is easy to establish that the entry nodes of the sub-graphs produced by the two algorithms for the same bytecode segment are related by $R$, and hence the result.

The proof proceeds by case analysis on the type of the producer instruction of the bytecode segment $\text{seg}_{JBC}(m,p)$. We present one interesting case in full detail; the full proof is presented in [29].

**Case $i = \text{athrow}$**

Let $X$ be the set containing the static type of the exception being thrown, and all of its sub-types. This set is the same for the direct and indirect extraction algorithms. Let $x \in X$.

The direct extraction for the $\text{athrow}$ instruction produces two edges, with the target node of the second edge depending on whether the exception $x$ is caught within the same method it was raised or not:

$$m\mathcal{G}((p, \text{athrow}), H) = \begin{cases} \{ \circ p \xrightarrow{\text{athrow}} p \xrightarrow{\text{handle}} q \} & \text{if has handler} \\ \{ \circ p \xrightarrow{\text{athrow}} p \xrightarrow{\text{handle}} p \} & \text{otherwise} \end{cases}$$
The transformation $\text{BC2BIR}_{\text{instr}}$ returns a single instruction. Then, similarly to $\mathfrak{mG}$, the $\mathfrak{bG}$ function produces two edges:

$$\text{BC2BIR}_{\text{instr}}(p, \text{athrow}) = \{\text{[throw } x]\}$$

$$\mathfrak{bG}([\text{throw } x]_{pc}, \vec{H}) = \begin{cases} 
\{ \circ c_{pc} \xrightarrow{x} \bullet p_{pc,x} \mapsto \circ p_{pc}, \bullet p_{pc,x} \mapsto \circ p_{pc} \} & \text{if has handler} \\
\{ \circ c_{pc} \xrightarrow{x} \bullet p_{pc,x} \mapsto \circ p_{pc}, \bullet p_{pc,x} \mapsto \circ p_{pc,x,r} \} & \text{otherwise}
\end{cases}$$

We have that $(\circ p_{pc}, \circ p_{pc}) \in R$. The transition $\circ p_{pc} \xrightarrow{x} \bullet p_{pc,x}$ is matched by the corresponding weak transition $\circ p_{pc} \Longrightarrow \bullet p_{pc,x}$. Thus obviously also $(\bullet p_{pc,x}, \bullet p_{pc,x}) \in R$. Next, there are two possibilities for the remaining transitions, depending on whether there is an exception handler for $x$ in $p$ and $pc$. If there is a handler, then we get

$$\bullet p_{m,x} \xrightarrow{\text{handle}} \circ q_{m}, \bullet p_{pc,x} \xrightarrow{\text{handle}} \circ p_{pc}, \text{ and clearly also } (\circ q_{m}, \circ p_{pc}) \in R.$$ 

If there is no exception handler for $x$, we get

$$\bullet p_{m,x} \xrightarrow{\text{handle}} \bullet p_{m,x,r}, \bullet p_{pc,x} \xrightarrow{\text{handle}} \bullet p_{pc,x,r}, \text{ and also } (\bullet p_{m,x,r}, \bullet p_{pc,x,r}) \in R.$$ 

This concludes the case. \qed
Chapter 4

Extraction of CFGs from Open Systems

We have presented in Section 2.1 a formal JVM framework, and in Section 3 we described a sound CFG extraction algorithm, which is proven to preserve the JVM behavior. Those definitions are valid for closed systems only. That is, they are applicable only for software systems whose all components are provided. However, there are scenarios where at least one of the components is not available, but still one wants to analyze the available components. Some common situations are systems under development, or a software system which depends on a third-part component. In both scenarios one may still extract CFGs and verify properties from the available components. The compositional verification technique presented in Section 2.5 could be used, for instance.

In this section we generalize the previous definitions to open Java bytecode systems. First, we extend the formal JVM framework to represent unavailable software components. The missing components are represented by user-provided interfaces. Next, we generalize the previous algorithm to modularly extract CFGs from the available components. That is, the algorithm extracts the method graphs for the available methods, and resolve the inter-dependencies involving missing methods by using the provided interfaces.

Eventually the missing components will arrive, and the open system will become close, and can execute. However, the arrival of a component may affect the soundness of CFGs which have been extracted previously. Thus, we define the refinement relation, which are constraints over the arrival of components, to guarantee that soundness is preserved. We conclude by proving that if the refinement relation holds, then the CFGs extracted with the modular algorithm from an open system are sound over-approximations for any closed system assembled from it. Therefore, the safety properties verified over the CFGs still hold.
CHAPTER 4. EXTRACTION OF CFGS FROM OPEN SYSTEMS

4.1 Open Java bytecode systems

Software systems are typically created by the assembly of several components. Each
software component provides specialized functionalities to other components, and
defines how the interaction occurs by providing an interface. We call an open system
any software system which has all its components defined, but at least one of the
component’s implementation is missing. In the current work we are interested
in extracting program models from the available software components of a Java
program. We accomplish this by generalizing the notion of an environment $\Gamma$ to
open Java bytecode systems.

$$
\Gamma_o^I : \text{Interface-Name} \rightarrow \left\{ \text{interfaces} : \text{set of Interface-Name}
, \text{method} : \text{set of Interface-Method-Ref} \right\}
$$

$$
\Gamma_o^C : \text{Class-Name} \rightarrow \left\{ \text{super} : \text{Class-Name} \cup \{\text{None}\}
, \text{interfaces} : \text{set of Interface-Name},
\text{fields} : \text{set of Field-Ref} \right\}
$$

$$
\Gamma_o^M : \text{Method-Ref} \rightarrow \left\{ \text{code} : \text{Instruction}^*,
\text{handlers} : \text{Handler}^* \right\}
$$

$$
\Gamma_o = \Gamma_o^I \cup \Gamma_o^C \cup \Gamma_o^M
$$

Figure 4.1: Definition of a Java bytecode open environment

Figure 4.1 shows the definition of open environment for Java bytecode, $\Gamma_o$. Most
of the definition looks similar to Definition 2.3 for closed programs. For instance, $\Gamma_o$
is defined as the union of the partial mappings from names to its classes, interfaces
and methods. Also, the mappings $\Gamma_o^I$ and $\Gamma_o^C$ are defined exactly as $\Gamma^I$ and $\Gamma^C$,
respectively.

The most notable difference is in the partial mapping from names to method
implementations, $\Gamma_o^M$. First, code array allows the existence of methods with empty
bodies, representing the missing components. For such methods, the definition
of $\Gamma_o^M[m].\text{handlers}$ has a different interpretation than $\Gamma^M[m].\text{handlers}$. Each
handler determines one exception type that can never be propagated by the missing
method, when it becomes available.

Open java bytecode systems are not executable, since at least one part is missing.
At some point the missing components will become available, then the system
becomes closed, and can be executed. Nevertheless, the goal of reasoning about
open environments is to be able to analyze the available code, and produce results
which hold for any closed system assembled from the open system. Thus, we need
to set constraints for the missing components, so that properties inferred from the
4.1. OPEN JAVA BYTECODE SYSTEMS

original open system are preserved once the system becomes closed. The arrival of a new component defines a new open environment, defined by the previous components, plus the newly added component. Thus, we define the constraints for property preservation as a refinement relation between the previous and the new open environments.

**Definition 11 (Refinement of open environments).** We define the refinement relation $\preceq$ between two open Java bytecode environments by the rules in Figure 4.2. Let $\Gamma_o$ and $\Gamma_o'$ be two open Java bytecode environments. We say that $\Gamma_o$ refines $\Gamma_o'$ if:

$\Gamma_o \preceq \Gamma_o'$

A closed environment is a special case of an open environment, where all method bodies are provided. We say that $\Gamma$ implements $\Gamma_o$ if $\Gamma \preceq \Gamma_o$, and $|\Gamma^M[m].\text{code}| > 0$ for all methods $m$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma^I_o[\omega].\text{interfaces} \subseteq \Gamma^I_o'[\omega].\text{interfaces}$</td>
<td>$\Gamma^I_o[\omega] \subseteq \Gamma^I_o'[\omega]$</td>
</tr>
<tr>
<td>$\Gamma^I_o[\omega].\text{methods} = \Gamma^I_o'[\omega].\text{methods}$</td>
<td>$\Gamma^I_o[\omega] = \Gamma^I_o'[\omega]$</td>
</tr>
<tr>
<td>$\Gamma^C_o[\sigma].\text{super} = \Gamma^C_o'[\sigma].\text{super}$</td>
<td>$\Gamma^C_o[\sigma] = \Gamma^C_o'[\sigma]$</td>
</tr>
<tr>
<td>$\Gamma^C_o[\sigma].\text{interfaces} \subseteq \Gamma^C_o'[\sigma].\text{interfaces}$</td>
<td>$\Gamma^C_o[\sigma] \subseteq \Gamma^C_o'[\sigma]$</td>
</tr>
<tr>
<td>$\Gamma^C_o[\sigma].\text{fields} \supseteq \Gamma^C_o'[\sigma].\text{fields}$</td>
<td>$\Gamma^C_o[\sigma] \supseteq \Gamma^C_o'[\sigma]$</td>
</tr>
<tr>
<td>$\Gamma^M_o[m].\text{handlers} \supseteq \Gamma^M_o'[m].\text{handlers}$</td>
<td>$\Gamma^M_o[m] \supseteq \Gamma^M_o'[m]$</td>
</tr>
<tr>
<td>$\Gamma^M_o'[m].\text{code} = \Gamma^M_o'[m].\text{code}$</td>
<td>if both methods are empty</td>
</tr>
<tr>
<td>$\Gamma^M_o[m].\text{handlers} = \Gamma^M_o'[m].\text{handlers}$</td>
<td>if both methods are concrete</td>
</tr>
<tr>
<td>$\text{EXCP}(\Gamma^M_o[m]) \cap \Gamma^M_o'[m].\text{handlers} = \emptyset$</td>
<td>if method is implemented</td>
</tr>
</tbody>
</table>

$\Gamma_o \preceq \Gamma_o' \iff \Gamma_o \preceq \Gamma_o'$

Figure 4.2: Refinement rules between open environments
same. This constraint is necessary to ensure that the open environments are defined for the same set of components.

An interface $\Gamma_{I}^o[w]$ refines $\Gamma_{I}^o[w]$ if both environments provide the same methods, and if the new interface inherits a subset of the interfaces inherited by the old one. These requirements are essential to guarantee that the virtual method call resolution produces a sound over-approximation of the possible call receivers. A class $\Gamma_{C}^o[\sigma]$ refines $\Gamma_{C}^o[\sigma]$ if it fulfills three requirements. First, the super class must be the same in both environments. The second requirement is that $\Gamma_{C}^o[\sigma]$ implements a subset of the interfaces implemented by $\Gamma_{C}^o[\sigma]$. This is a relaxation from the definitions, and it is allowed because implementing fewer interfaces means that the original class is an over-approximation of the refined class. The last requirement is that the fields defined in the refined class is a superset of the fields in the original class. Intuitively, any analysis performed in the original classes accounted all the fields required by it. Thus, if the class will contain any new field afterwards, they were not referenced in the original class, and do not compromise the analysis.

The refinement between two methods $\Gamma_{M}^o[m]$ and $\Gamma_{M}^o'[m]$ is defined for three distinct cases. The first is between two methods with empty bodies. In this case, $\Gamma_{M}^o[m]$ is said to refine $\Gamma_{M}^o'[m]$ if the set of exceptions that the former guarantees not to propagate is a superset of the exceptions not propagated by the original method. That is, the refinement restricts the set of exceptions that a method may propagate. The second and simplest case, is the refinement between two concrete methods, i.e., without empty bodies: both $\Gamma_{M}^o[m]$ and $\Gamma_{M}^o'[m]$ must have the same code and handlers. The third, and most relevant case, is the one where an empty method $\Gamma_{M}^o'[m]$ is refined by a concrete method $\Gamma_{M}^o[m]$. Let $EXCP$ be the auxiliary function which returns the set of exceptions that some method $\Gamma_{M}^o[m]$ propagates. The only constraint is that the refined method cannot propagate any of the exceptions declared to be caught and handled in the original method.

One important property following the definition of open environments for Java bytecode is that the refinement relation preserves the type hierarchy. This means that the sub-typing relation between classes is the same, and the sub-typing relation between interfaces, and interfaces and classes in the original open environment is an over-approximation from the hierarchy in the refined model. The definition below presents the class hierarchy for a Java program.

**Definition 12 (Type Hierarchy).** Let $\Gamma_{I}^o$ and $\Gamma_{C}^o$ be partial maps from interface and classes names, respectively, to theirs attributes. The Java type hierarchy is defined as $\mathcal{L} = \Gamma_{C}^o \cup \Gamma_{I}^o$.

Another consequence of the type hierarchy preservation is that the set of exception types is also known. As mentioned in Section 2.1, an exception in a Java program is an object whose class is a sub-type of the `java.lang.Throwable` class. Since the class type definitions is the same for both the original environment, and for the open environments that refine it, we can estimate the set of exceptions types. The following definition presents the set of exception types.
4.2. THE \( \text{oG} \) EXTRACTION ALGORITHM

Definition 13 (Exceptions Set). Let \( \mathcal{L} \) be a given class hierarchy. We define \( \mathcal{ANY} = \{ c \in \mathcal{L} | c < C: \text{java.lang.Throwable} \} \), which represent the exceptions type in a given open environment.

4.2 The \( \text{oG} \) Extraction Algorithm

In this section we present a modular CFG extraction algorithm that receives an open environment as input, and produces CFGs for its available components. It generalizes the \( \text{mG} \) algorithm for closed programs only, presented in Section 3.

The extraction is made using the transformation into the BIR language, following the initial proposal in [2]. Once again, the motivation to use BIR is its support for exceptions. The transformation from JBC to BIR is purely syntactic. Thus the mapping between an open Java bytecode environment into an open BIR environment is trivial: the partial maps \( \Gamma^I_o \) and \( \Gamma^C_o \) are the same for both environments, and the mapping \( \Gamma^M_o \) is the result of the transformation \( \text{BC2BIR} \) for each available method.

Let Method-Ref be the set of all method references, and \( \text{ENV}_{BIR} \) be the set of all open BIR environments. The define the transformation function \( \text{oG} \) as follows.

Definition 14 (Modular Extraction Function). Let \( \Gamma_o \) be an open environment. We define the method graph extraction function of a method \( \Gamma_o \models \Gamma_o[m] \), where \( |\Gamma_o[m].\text{code}| > 0 \), as \( \text{oG} : \text{Method-Ref} \times \text{ENV}_{BIR} \rightarrow (V \times L \times V)^* \). The definition of instruction-wise rules is presented in Figure 4.3.

The extraction rules for CFGs are very similar to the ones presented in [2]. In fact, the rules for instructions that depend only on intra-procedural information (own method’s instructions and exception handler table) are the same in both algorithms. This is the case for all instructions, except the ones that execute method calls: \text{NewObject} and \text{MethodCall}. The analysis of method calls imposes two main challenges in a modular set-up: the resolution of virtual method calls, and the propagation of uncaught exceptions.

Virtual method calls happen only for the instructions in \text{MethodCall} which were originally \text{invokevirtual} and \text{invokeinterface} JBC instructions before the translation to BIR. In contrast to the \( \text{bG} \) algorithm for closed programs, \( \text{oG} \) cannot be parametrized by a virtual call resolution algorithm. Most of such algorithms, such as RTA or XTA, rely on code analysis to prune out spurious method calls. However, since we extract CFGs from open systems, any algorithm based on code analysis would be unsound. For example, one method could not be listed as a possible receiver to a virtual call because no object of its class type are referenced at the available code, but will when a missing component is defined.

Instead, we define a fixed algorithm for virtual method call resolution: \text{MCA}. This algorithm resolves virtual calls using only information from the type hierarchy. It receives as input a method signature \( \text{ns} \), and the object’s static type \( C \), and outputs as possible receivers all the methods from sub-types of \( C \) with the same
\[
MCA(\text{C.ns}) = \begin{cases} 
\{ \sigma.\text{ns} \mid \sigma \ll C \} & \text{if call is virtual} \\
\{ \text{C.ns} \} & \text{otherwise}
\end{cases}
\]

\[
\mathcal{H}_E = \begin{cases} 
\{ (\circ_{m}^{\text{x}}, \text{handle}, \circ_{m}^{\text{c'}}) \} & \text{if } h_{H}(\text{pc}, x) = \text{pc'} \neq 0 \\
\{ (\circ_{m}^{\text{x}}, \text{handle}, \circ_{m}^{\text{c'},x}) \} & \text{if } h_{H}(\text{pc}, x) = 0
\end{cases}
\]

\[
oG(i_{\text{pc}}, H) = \begin{cases} 
\{ (\circ_{m}^{\text{c}}, \varepsilon, \circ_{m}^{\text{c+1}}) \} & \text{if } i \in \text{Assignment} \cup \\
\{ (\circ_{m}^{\text{pc}}, \varepsilon, \circ_{m}^{\text{pc+1}}), (\circ_{m}^{\text{pc}}, \varepsilon, \circ_{m}^{\text{pc'}}) \} & \text{if } i = [\text{if expr pc'}] \\
\{ (\circ_{m}^{\text{c}}, \varepsilon, \circ_{m}^{\text{c+1}}) \} & \text{if } i = [\text{goto pc'}] \\
\{ (\circ_{m}^{\text{pc}}, \varepsilon, \circ_{m}^{\text{pc+1}}), (\circ_{m}^{\text{pc}}, \varepsilon, \circ_{m}^{\text{pc'},x}) \} & \text{if } i \in \text{Assertion} \\
\{ (\circ_{m}^{\text{pc}}, \varepsilon, \circ_{m}^{\text{pc+1}}), (\circ_{m}^{\text{pc}}, \varepsilon, \circ_{m}^{\text{pc'},x'}) \} & \text{if } i \in \text{NewObject} \\
\mathcal{H}_E \cup N^C & \text{if } i \in \text{MethodCall}
\end{cases}
\]

\[
N^E = \begin{cases} 
\bigcup_{x \in \text{G}(\text{n})} \{ (\circ_{m}^{\text{pc'}}, \text{handle}, \circ_{m}^{\text{pc, x'}}) \} & \text{if } |\Gamma_{\text{m}}[\text{n}].\text{code}| > 0 \\
\mathcal{H}_E & \text{otherwise}
\end{cases}
\]

Figure 4.3: Modular extraction rules for control flow graphs from BIR

signature. This is shown as the first case of the MCA function in Figure 4.3. The second case, for non-virtual calls, can return a single receiver.

The function \( \mathcal{N} \) extracts the edges caused by exceptions propagated by a called method \( n \). The definition of \( \mathcal{N} \) contains two cases: the first one is when the called method is implemented in the current open environment \( \Gamma_{\circ} \). Here, we proceed in the same fashion as with the analysis of closed programs: check if the called method’s control flow graph contains exceptional return nodes, denoting uncaught exceptions, and add correspondent edges, depending on the presence of a suitable exception handler.

The second case is when the called method is missing. The analysis of exceptional flow in this scenario is more complex because the set of exceptions that a method can raise depends on the instructions from the method. Thus it is impossible to determine in advance the exact exception types that a missing method may propagate. One could suggest that the user guarantees by contract the set of possible exceptions a method can raise, and verify such set upon the arrival of the missing method. However, this approach is too restrictive, because it forces the user to annotate the propagated exceptions set even before the method
4.3 Correctness Proof

In our approach we are concerned with the soundness of the analysis. Thus, we must conservatively over-approximate the set of propagated exceptions by a missing method. First, we assume this to be the set $\mathcal{AN} \cup \mathcal{Y}$ of all exception types, that is, all subtypes of `java.lang.Throwable`. Also, the user may provide a set of exceptions that the missing method will never propagate, defined in $\Gamma_o^m[m].\text{handlers}$. Therefore, we denote the set of the exceptions which a missing method can propagate as $\mathcal{AN} \cup \mathcal{Y} - \Gamma_o^m[m].\text{handlers}$.

4.3 Correctness Proof

In this section we enunciate the correctness proof for the modular extraction of CFGs. First, we show that if an open environment refines another, it implies that the control flow graph extracted with $oG$ from a method of the original environment is a superset of the CFG of the same method for the refined environment. That is, we show that the refinement relation is monotone with respect to set inclusion, and by consequence, also to structural simulation.

As mentioned in Section 4.2, a closed Java program is a special case of an open environment, where there are no missing methods. Thus, the CFGs extracted modularity from an open environment are a superset of the CFGs from any closed system that implements the open environment. Next, we show that the CFGs extracted from closed systems using the modular extraction algorithm, and those extracted using the algorithm for closed programs, are the same.

A previous result [3] has proven that the CFG behavior induced by CFGs extracted with $bG$ simulate the JVM behavior. Another previous result [15] states that structural simulation implies behavioral simulation. Thus, we conclude that also the CFGs extracted modularly preserve the behavior. Therefore, the verification is sound for any class of properties preserved by simulation.

Now we enunciate the monotonicity of the refinement relation w.r.t the set inclusion of CFGs, and consequently to structural simulation.

**Theorem 2** (Containment of CFGs). Let $\Gamma_o$ and $\Gamma_o'$ be two open environments, and $m$ be some method signature such that $|\Gamma_o^M[m].\text{code}| \neq 0$ and $|\Gamma_o'[m].\text{code}| > 0$. Then $\Gamma_o \preceq \Gamma_o'$ implies that $oG(m, \Gamma_o) \subseteq oG(m, \Gamma_o')$.

**Proof.** The proof goes by case analysis on the BIR instructions set.

By the hypothesis, the method $m$ must be implemented in both environments. Also by the hypothesis, $\Gamma_o \preceq \Gamma_o'$ holds. Thus, from the refinement definition (Fig 4.2), the instructions array and exception table are the same. ($\Gamma_o^M[m].\text{code} = \Gamma_o'[m].\text{code}$ and $\Gamma_o^M[m].\text{handler} = \Gamma_o'[m].\text{handler}$).

It is trivial to see from the definition of function $oG$ (Fig. 4.3 on page 38) that it outputs the same set of triples for all instruction groups whenever two methods have the same instructions array and exception table, except for `NewObject` and `MethodCall`. Thus, the trivial cases are proven, and there are only two cases left.
The sets *NewObject* and *MethodCall* contain instructions that execute method invocations. The former contains the instructions that invoke object constructors. The later set contains the other method invocation instructions, either virtual or non-virtual. The non-virtual method calls, including constructor calls, have only one possible receiver. The possible receivers for a virtual method call, however, depend on the class hierarchy *L* of each open environment. Moreover, the computation of *N^pec* also depends on the definitions of the open environment. Therefore, we present the proof for the virtual case for the *MethodCall*. The other are special cases, where there is a single receiver for the call, and the proof is analogous.

The function *oG* produces a set of varying size for each possible receiver *n* of a virtual method call (*n* ∈ *MCA(C.ns)*): one normal edge, denoting a successful return from *n*, plus pairs of edges for each exception that *n* may propagate (by function *N^pec*). First, we show that the set of receivers for a virtual method call in *Γ_o[m].code* is a subset of the receivers for the same call in *Γ_o'[m].code*. Then, we conclude by showing that the set of propagated exceptions for a call to method *n* is always a subset in the refined environment.

Let *MCA_L(C.ns)* and *MCA_L(C.ns)* be the sets of all possible receivers of the same virtual method call in the original and refined open environments, respectively. They are defined as all the methods from the sub-type *C* with the same signature *ns*. The static type *C* can be either an interface, or a class. First, let *C* be a class. The refinement relation defines that all classes in both *Γ^C_o* and *Γ^C_o'* have the same super class. The subtyping relation (Fig. 2.5, page 13) defines that classes can only be subtyped by another class. Thus, the set of sub-types of *C* is the same in both environments, and *MCA_L(C.ns) = MCA_L(C.ns)*. Now let *C* be an interface. Its sub-types are all the classes (and their sub-types) that implement the interface, or one of its sub-interfaces. The refinement relation defines that both the classes that implement an interface (*Γ^C^o_o'[σ].interfaces*) and its sub-interfaces (*Γ^C^o_o'[ω].interfaces*), must be subsets in the refined open environment. Thus, the subtypes of *C* in the refined environment are a subset of the original environment. Therefore, *MCA_L(C.ns) ⊆ MCA_L(C.ns)*.

Next, we show that the set of propagated exceptions by an arbitrary receiver of a method call in *Γ^M_o'[m].code* is a superset of the exceptions propagated by the same call in *Γ^M_o[m].code*. Let *Γ^M_o[m]* be one possible receiver for a method invocation within *Γ^M_o'[m]*. The refinement relation defines three cases for the method.

The first case is when both *Γ_o[n]* and *Γ_o'[n]* are empty methods. In this case, the set of exceptions guaranteed never to be propagated by the method in the refined environment must be a superset of the exceptions never propagated in the original version. The function *N^pec* falls always into the second case, since the method implementation is missing in both open environments. The set of edges produced by this function is inversely monotone to the number of exceptions declared to be caught by the missing method (*AN^L_o' = Γ^M_o'[n].handlers*). Therefore, the number of edges produced by *N^pec* for the refined environment must be a subset of the edges produced for the original environment, since *Γ^M_o[n].handlers ⊆ Γ^M_o'[n].handlers*.

In the second case the method *Γ_o'[n]* is implemented by *Γ_o[n]*. The refinement
4.3. CORRECTNESS PROOF

relation constrains the set of exceptions that the method implementation may propagate. It cannot contain an exception that the method declares not to propagate in the original environment. Therefore, if the refinement relation holds, the set of exceptions propagated by the refined method is clearly a subset of the exceptions propagated by the method from the original environment.

The third case is when $\Gamma_o[n]$ and $\Gamma_o'[n]$ are implemented methods. In this case, there is no constraint over the exceptions it may propagate. However, since both the code and handlers are preserved, the only possible difference in the CFG is in the case that a method called within $\Gamma_o[n]$ has been implemented, or is still a missing method. Thus, by the two previous cases, the set of propagated exceptions in the refined environment still has to be a subset of the original environment.

We conclude that the set of possible receivers for a virtual method call, and the set of propagated exceptions for a method invocation, are both a subset in the refined environment, when compared to the original environment.

As mentioned above, a closed system is a special case of an open system, where there are no missing components. Thus, it is possible to extract CFGs from closed programs using the modular algorithm. However, we want to show that the algorithm is precise, and that it over-approximates only whenever it is necessary. We do this by showing that the control flow graphs extracted by the modular algorithm $oG$ are the same as the one extracted by $bG$ using the MCA algorithm for virtual method call resolution. This result is enunciated in the following proposition.

**Proposition 1** (Modular Control Flow Graph). Let $\Gamma_o$ be a closed environment, and $bG_{MCA}$ be the implementation of $bG$ using the MCA algorithm for virtual method call resolution. Then $bG(\Gamma_o[n]) = oG(\Gamma_o[n])$.

*Proof.* The proof follows from the fact that if there are no missing components, then the function $N_{\pi n}^e$ always falls into the first case. Thus, its definition is the same for both extraction algorithms. Also, the MCA algorithm outputs the same set of virtual method call receivers. Therefore, the extraction rules for $oG$ are reduced to exactly the same extraction rules as for $bG$. 

\[\square\]
Chapter 5

The ConFlEx Tool

The algorithms presented in Sections 3.1 and 4.2 have been implemented in the ConFlEx tool. In the following sections we describe the implementation aspects, and present practical results.

5.1 Implementation

This section describes the Control Flow graph Extractor tool (ConFlEx) \cite{10}, which implements the two CFG extraction algorithms: \( bG \) (Def.10), and \( oG \) (Def.14).

The tool is written in OCaml, and uses Sawja \cite{5, 17}, a library for the static analysis of Java bytecode programs. Sawja provides high-level functions to manipulate bytecode .class files, implements algorithms for virtual method call resolution, and transformations from bytecode into intermediate representations.

We have tailored Sawja in several parts. The major enhancement was on the BC2BIR transformation, to provide an accurate estimation of the possible exception types raised by the BIR instruction [throw]. In the standard implementation of Sawja, the BC2BIR transformation does not consider the program’s class hierarchy. It only performs a syntactic transformation, and associates \texttt{java.lang.Object} to expressions and variables of non-primitive types. We altered the symbolic execution of BC2BIR to associate types to variables and expressions. Also, in operations involving non-primitive types, we compute the type as the common super-type between the operands. This is a conservative estimation of the actual type, but still sound for modular set-ups.

We have implemented the formal definitions that we have introduced as new modules in Sawja. The module DefCFG represents and manipulates CFGs, as presented in the Definitions 3 and 5. We wrote it as a separate module to make ConFlEx independent from a single CFG definition. This facilitates in the future to add other CFGs definitions. The OpenEnv module implements the representation of an open Java bytecode system as an open environment, following the definitions in Figure 4.1. The environment is represented by a structure containing three
CHAPTER 5. THE CONFLEx TOOL

associative maps, representing $\Gamma^I_0$, $\Gamma^C_0$, and $\Gamma^M_0$. The module also implements the check of the refinement relation, following the rules in Figure 4.2. The module MCA implements the virtual method call resolution algorithm for open systems. The algorithm over-approximates the set of possible receivers to a virtual call as presented in Section 4.2: the set of all methods with same signature, and from a subclass, of the invocation instruction’s operand. We have adapted the code of the Class Reachability Analysis (CRA), native from Sawja, to implement MCA.

The interfaces for the missing components are provided as a combination of Java annotations \cite{32}, and dummy methods containing a single return instruction. The use of Java annotations allows us to set the granularity of components to method-level. Moreover, Sawja conveniently provides built-in support for the manipulation of Java annotations. The dummy methods are necessary because the annotations must be associated to a method in the .class file.

```java
import java.lang.annotation.*;

@Retention(value = RetentionPolicy.CLASS)
@Target(value = {ElementType.CONSTRUCTOR,ElementType.METHOD})
public @interface GhostComponent
{
    String[] req_meths();
    String[] handlers();
}
```

Figure 5.1: The GhostComponent annotation

We have defined the GhostComponent annotation template, containing the mandatory annotations for the missing methods. The user must compile a .class file with the code in Figure 5.1 and put it into each directory containing an annotated method. The user annotates a method by adding a custom GhostComponent annotation immediately before the declaration of the dummy method. The field req_meths is an array of strings specifying the methods declared to be invoked in the method code. The field handlers is an array of strings specifying which exceptions the user declares that the missing method will never propagate when its code becomes available. The standard built-in annotations Retention and Target defines the levels of visibility and granularity of the annotation, respectively. The former states that the annotation is only visible in the class file, but not at runtime; this suffices for our purposes. The latter determines that the elements which can be annotated are methods, both standard and constructor. Figure 5.2 shows an example of the source code of an annotated missing method.

By default, CONFLEx does not consider methods from the standard Java library (API) to be part of the program, and it does not extract their CFGs. The tool considers only the client program, which is the code that the user explicitly declares.
5.1. IMPLEMENTATION

@GhostComponent(
    req_meths = {
        "java.lang.String packA.classB.metC(int[],packD.classE)"
    },
    handlers = {
        "java.lang.NullPointerException",
        "java.lang.ArithmeticException",
        "UserDefinedExceptionA",
    }
)
public String myExampleMissing(String a) {
    return a;
}

Figure 5.2: Example of an annotated dummy method

to belong to the program. The client code must contain an entry method main. ConFlEx assumes that calls to methods from the API are side-effect free. That is, there cannot be call-backs, which are calls within the API methods to method from the client program. Also, we assume that API methods can only propagate exceptions declared in the throws field of the method’s signature. The user can alter ConFlEx’s assumptions w.r.t to API methods by explicitly declaring the standard API to be a component of the program. Unfortunately the Java standard API is large, and the consequence are unnecessarily large CFGs. That is the reason behind the design choice to extract only the client code by default.

We have introduced in Section 4.1 the set $\mathcal{ANY}$ of all exception types in an open environment, and have presented it in Definition 13 as all the subtypes of java.lang.Throwable. In practice, we can filter out exception types which can not be raised because they are not referenced neither in the code, nor in the interfaces. We compute the $\mathcal{ANY}$ as the union of: (a) exceptions represented by the BIR assertions (Fig. 2.11); (b) user-defined exception classes; and (c) exceptions declared as potentially throwable by the API methods.

ConFlEx implements the caching of the intra-procedural analysis. The caching has two benefits. First, it allows to check if the refinement relation holds between two versions of the program. Second, it allows the incremental extraction of newly provided components, in contrast to an entire new intra-procedural analysis. This is valid even if an open environment is not a refinement of another, as long as ConFlEx is executed with the same configuration options. The feature exploits the fact that the computation of the intra-analysis is preserved if the implementation of a method is not altered. Still, ConFlEx recomputes the entire inter-procedural analysis, so that the control flow caused by propagated exceptions is over-approximated the least as possible.


5.2 Experimental Results

We have tested ConFlEx on three different programs: Jasmin version 2.4 [27]; JavaCUP version 11a beta [18]; JFlex version 1.4.3 [25]. All the tests have been made on a machine with an Intel i3 2.27 GHz processor and 4GB of RAM. We have considered the seven scenarios presented in Table 5.1. They reflect the incremental refinements we have performed on each program. Specifically, by performing the first refinement over the initial open system (Scenario 1), we have the Scenarios 2 and 3. Also, the Scenario 4 represents the implementation of the resulting open system from Scenario 3 into a closed environment. There is no refinement in Scenarios 5, 6, and 7, and the extraction of CFGs is performed over the original closed programs.

We have artificially generated the missing methods by replacing the actual implementations with dummy methods, and GhostComponent annotations. The req_meths field was annotated with the methods that are called in the actual method implementation. The handler field was empty in all cases. That is, we considered that the missing methods could potentially propagate any exception in the ANY set. The missing classes in Scenario 1 are: ClassFile, InsnInfo, Scanner and parser for Jasmin; NFA, SemCheck, RegExp and IntCharSet for JFlex; symbol_set, terminal_set, lalr_state and production for JavaCUP. The missing classes in Scenarios 2 and 3 are: Scanner for Jasmin; NFA for JFlex; production for JavaCUP.
Table 5.2 shows the experimental results. The considered data are: number of JBC and BIR instructions; number of nodes and edges of the CFG after the inter-procedural analysis; the parsing time; time of the intra-procedural analysis; and time for the inter-procedural analysis.

The parsing time is the time used by Sawja for VMC resolution, and to transform JBC into BIR. The extraction of CFGs from BIR is divided into two stages. First, the intra-procedural analysis extracts control flow graphs for each BIR method by applying the formal rules in Figure 3.1 except the function $\bar{N}$. This function extracts the transitions related to exceptions that are propagated from called methods. As described in Section 3.1, this is computed in the inter-procedural analysis, using a fixpoint algorithm similar to the one of Jo and Chang [24].

<table>
<thead>
<tr>
<th>Scen.</th>
<th># of JBC Instrs.</th>
<th># of BIR Instrs.</th>
<th># of Nodes</th>
<th># of Edges</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Parsing INTRA INTER</td>
</tr>
<tr>
<td>Jasmin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>25318</td>
<td>8765</td>
<td>46202</td>
<td>46960</td>
<td>2756</td>
</tr>
<tr>
<td>2</td>
<td>30255</td>
<td>10727</td>
<td>27695</td>
<td>28084</td>
<td>2672</td>
</tr>
<tr>
<td>3</td>
<td>30255</td>
<td>10727</td>
<td>27695</td>
<td>28084</td>
<td>2652</td>
</tr>
<tr>
<td>4</td>
<td>32101</td>
<td>11301</td>
<td>25842</td>
<td>26243</td>
<td>2620</td>
</tr>
<tr>
<td>5</td>
<td>32101</td>
<td>11301</td>
<td>25842</td>
<td>26243</td>
<td>2636</td>
</tr>
<tr>
<td>6</td>
<td>32101</td>
<td>11301</td>
<td>25842</td>
<td>26243</td>
<td>1960</td>
</tr>
<tr>
<td>7</td>
<td>32101</td>
<td>11301</td>
<td>23651</td>
<td>23996</td>
<td>844</td>
</tr>
</tbody>
</table>

| JavaCUP |                 |                  |            |            |           |
| 1     | 28397            | 11616            | 57636      | 58048      | 2292      | 712       | 212       |
| 2     | 31709            | 13030            | 58183      | 58612      | 2244      | 280       | 720       |
| 3     | 31709            | 13030            | 58183      | 58612      | 2276      | 884       | 176       |
| 4     | 33777            | 13958            | 28918      | 29197      | 2228      | 24        | 404       |
| 5     | 33777            | 13958            | 28918      | 29197      | 2384      | 868       | 16        |
| 6     | 33777            | 13958            | 28918      | 29197      | 2088      | 692       | 12        |
| 7     | 33777            | 13958            | 26714      | 27038      | 1120      | 686       | 17        |

| JFlex |                 |                  |            |            |           |
| 1     | 49262            | 19238            | 84257      | 85675      | 8433      | 3116      | 260       |
| 2     | 52898            | 20479            | 56699      | 58167      | 8121      | 0640      | 1292      |
| 3     | 52898            | 20479            | 56699      | 58167      | 8077      | 3332      | 84        |
| 4     | 57343            | 22117            | 48279      | 49058      | 8049      | 96        | 1168      |
| 5     | 57343            | 22117            | 48279      | 49058      | 8009      | 3400      | 20        |
| 6     | 57343            | 22117            | 48279      | 49058      | 6756      | 1344      | 36        |
| 7     | 57343            | 22117            | 46442      | 47072      | 2892      | 1302      | 39        |

Table 5.2: Experimental results for ConFlEx

The number of BIR instructions is less than 40% of bytecode instructions, for all cases. This indicates that the use of BIR alleviates the blow-up of CFGs, and
clearly the program analysis benefits from this.

The number of instructions (both JBC and BIR) increases whenever a new component is fully provided. Obviously, this happens because the incoming methods sum in the count. Surprisingly, in some cases, such as the Scenarios 1 and 2 for Jasmin, the number of nodes is larger when more classes are missing, and consequently there are less instructions. This is caused by the over-approximation of the exceptional control flow, and the consequent addition of nodes tagged with exceptions. However, this is not always the case, as it can be seen in the refinement from Scenario 1 to either Scenario 2 or 3 in the analysis of JavaCUP. This happens because the size of the over-approximations depends on the number of the calls to missing methods. If there are few calls, then there are fewer over-approximations to perform.

We can see that, on average, the computation times for intra- and inter-procedural analyses grow linearly w.r.t number of BIR instructions. However, this growth depends heavily on the number of exceptional paths in the analyzed program. As expected, the time to perform the intra-procedural analysis decreases considerably whenever we reuse the cache of a previous analysis, instead of computing a whole new analysis for an open system. Surely this follows from the fact that only newly provided method are extracted, and typically this number is much lower than the total number of methods. However, the caching of analyses is still a feature under development. We have noticed that the time for loading the cache is significant. This is caused by the use of XML to store the analyses, and the enormous files it creates, and parses. We are already studying alternatives to replace the use of XML files by a more compact and faster storage medium.

A strange behavior has been detected in the performance of the inter-procedural analysis. We expected similar times for similar scenarios, such as Scenarios 2 and 3, and Scenarios 4 and 5. However, the inter-procedural analysis is taking longer when using the cached analyses. The analyses are coherent though, since we get the same CFGs in both cases. After several discussions, we have concluded that the most likely reason for the time difference is because the OCaml’s garbage is being executed right before the inter-procedural analysis, when the reusing the cache. This is plausible because loading and parsing the enormous XML files consume a lot of memory, which can be freed right after the intra-procedural analysis. The verification of the hypothesis is pending, and is our most immediate task. However, if confirmed, we automatically fix it by replacing the usage of XML.

We have executed ConFlEx using MCA, CRA and RTA over the complete programs, to compare the precision and efficiency of both algorithms for closed and open systems. Since CRA is a refinement of MCA, we get shorter times to parse the program and to compute the intra- and the inter-procedural analyses. However, the performance using RTA was far superior, with parsing times always less than half of the parsing time of CRA. Moreover, the RTA showed a considerable precision when compared to CRA, and the size of its CFGs were 10% smaller.

To conclude, we do not provide comparative data with other extraction tools, such as Soot [44], or Wala [22] because this would demand the implementation of
similar extraction rules from their intermediate representations. However, experimental results from Sawja [17] show that it outperforms Soot in all tests w.r.t. the transformation into their respective intermediate representations, and outperforms Wala w.r.t. virtual method call algorithms. Thus, our ConFlEx clearly benefits from using Sawja and BIR. Also, to the best of our knowledge, ConFlEx is the first control flow analysis tool that supports modular CFG extraction.
Chapter 6

Related work

Java bytecode has several aspects of an object-oriented language that make the extraction of control flow graphs complex, such as inheritance, exceptions, and virtual method calls. Therefore, in this section we discuss the work related to extracting CFGs from object-oriented languages. To the best of our knowledge, for none of the existing extraction algorithms a correctness proof has been provided.

Sinha et al. [35, 36] propose a control flow graph extraction algorithm for both Java source and bytecode, which takes into account explicit exceptions only. The algorithm performs first an intra-procedural analysis, computing the exceptional return nodes caused by uncaught exceptions. Next, it executes an inter-procedural analysis to compute exception propagation paths. This division is similar to how our algorithm analyses exceptional flows, using a slightly different inter-procedural analysis. However, the authors do not discuss how the static type of explicit exceptions is determined by the bytecode analysis, whereas we get this information from the BIR transformation. Moreover, the use of BIR allows us to also support (a subset of the) implicit exceptions.

Jiang et al. [23] extend the work of Sinha et al. to C++ source code. C++ has the same scheme of try-catch and exception propagation as Java, but without the finally blocks, or implicit exceptions. This work does not consider the exceptions types. Thus, it heavily over-approximates the possible flows by connecting the control points with explicit throw within a try block to all its catch blocks, and considering that any called method containing a throw may terminate exceptionally. Our work consider the exceptions types. Thus, it produces more refined CFGs, and also tells which exceptions can be raised, or propagated from method invocations.

Choi et al. [8] use an intermediate representation from the Jalapeño compiler [7] to extract CFGs with exceptional flows. The authors introduce a stack-less representation, using assertions to mark the possibility of an instruction raising an exception. This approach was followed by Demange et al. when defining BIR, and proving the correctness of the transformation from bytecode. As a result, our extraction algorithm, via BIR, is very similar to that of Choi. We differ by defining
formal extraction rules, and proving its correctness w.r.t. behavior.

Finally, Jo and Chang [24] construct CFGs from Java source code by computing normal and exceptional flows separately. An iterative fixed-point computation is then used to merge the exceptional and the normal control flow graphs. Our exception propagation computation follows their approach; however, the authors do not discuss how the exception type is determined. Also, only explicit exceptions are supported; in contrast, we determine the exception type and support implicit exceptions by using the BIR transformation.

Several algorithms about the resolution of virtual method calls exist in literature. The most popular are Class Hierarchy Analysis (CHA) [11, 43] and Rapid Type Analysis (RTA) [42, 13]. In the former, the sets of possible method call receivers are built by simply looking at the inheritance relations between the classes; in the latter, by considering the instantiation of objects. CHA is widely used because of its simplicity and performance, although it is not precise. RTA is remarked as a good trade-off between performance and accuracy of the results. Both algorithms rely on code analysis. Nevertheless, in a modular scenario a VMC must be resolved despite the absence of some components. Thus, we have introduced MCA in in Section 5.1 which is essentially a simplification of CHA, and over-approximate the receivers soundly by using the class hierarchy only.

Sundaresan et al. [42] present two techniques that exploit an intermediate representation of the bytecode, based on simple assignments, to determine the call receivers. These techniques work on an existing call graph. Their goal is to determine the reachable types for a receiver. Given a type, they check if there is an execution path between the instantiation of an object of the given type and the assignment of some variable of that type to the object, which becomes a possible receiver. This leads to the construction of a secondary graph, built upon the first, and called type propagation graph. The hierarchy is used at the end of the algorithm to filter the sets of possible receivers from the nodes in the graph. Our approach highly differs from theirs: we are interested in incomplete programs, whereas their results are presented as valid for closed systems only. Also, differently from their work, we build our own call graph from scratch, by using the class hierarchy only. We do not refer to any code analysis to build the set of receivers.

Tip and Palsberg [43] present four algorithms that compute the sets of receivers for all the call expressions along the program. These sets approximate the run-time value types that the expressions assume. All the sets are distinct for each class, method, field, according to the chosen algorithm. The main difference between the algorithms is the number of sets used to determine the type of a call receiver. This value influences the implementation and performance costs, as well as the accuracy of the results. It is shown that the fewer sets are defined, the more scalable is the algorithm. These algorithms depend on the analysis of the whole program code, so they are applicable for closed programs only. Our work present a strategy to analyze programs in a modular way instead, thus it also applies programs not provided completely. However, we use a variant of the CHA algorithm as described in [43]: we over-approximate the set of method call receivers by considering the class
hierarchy only. The class hierarchy is retrievable from the available components together with the information contained in the provided interfaces.

One of the goal of this work is the modular analysis of software systems. Modularity is hard to achieve because of the inter-dependencies existing among the system components. Cousot [9] presents four strategies to perform static analyses of programs in a modular fashion. The first is the simplification-based: in it, the inter-dependencies are simplified globally; this saves resources but is not efficient. The second is the worst-case, where no information whatsoever about inter-dependencies is assumed; this is very efficient but often too imprecise. The third is the user-provided interfaces: the user (manually or automatically) provides the information needed to resolve an inter-dependencies; this is done in terms of assumptions and guarantees. The last is the symbolic relational: symbolic names replace all the actual references within an inter-dependence relation; only when all components are available, the dependence effects are evaluated. The current work is classified as the third method, since we propose the use of interfaces to get knowledge about unavailable components. These interfaces have to express all the assumptions we can guarantee about them.

The work by Rountev [34] is an example of a modular program analysis, which targets incomplete programs. The author presents a technique to identify correctly side-effect-free methods in Java software. The approach is based on a class analysis, that determines the possible classes which an object reference points to. In our work, we assume that all the methods in the API libraries are side effects-free. Even though the described approach differs from ours, it might be used to prove or disprove our assumption. We could apply this technique preliminarily, before the extraction and analysis. Also, possible side effects might be added to the control flow graph extracted.
Chapter 7

Conclusion

Now we summarize the presented work, theoretical and experimental results, and point directions for future work.

7.1 Discussion

This thesis presents two variants of an algorithm for the extraction of CFGs from Java bytecode, which are sound w.r.t to the JVM behavior, and considers precise exceptional flow. The extracted CFGs from both algorithms are suitable for various control flow analyses, in particular for formal verification. The first algorithm is applicable only to closed systems, i.e., fully provided programs. It is defined indirectly: first, the JBC is transformed into BIR, an intermediate stack-less representation of bytecode; then the CFG is extracted from the BIR representation. The use of BIR is motivated by its precise information about exceptional control flow. We state and prove its soundness: the behavior of the extracted graphs simulates the behavior of the original programs. To the best of our knowledge, this is the first CFG extraction algorithm that has been proved correct. The proof is non-trivial, relying on several results to obtain a relatively economic correctness argument phrased in terms of structural simulation.

The second algorithm generalizes the first one to open systems. Even though an open Java bytecode program cannot execute, one may still want to analyze the available components. The compositional verification technique proposed by Gurov et al. [15] is an example of a technique that directly benefits from the presented work. We have extended a previous formal Java bytecode framework to represent open systems, where missing components are described by user-provided interfaces. Then, we generalize the previous indirect algorithm to extract CFGs from the available components of an open Java bytecode system. The inter-dependencies involving missing components are resolved using the information in the interfaces. The arrival of a missing component redefines the open system. To preserve previous results, we define a refinement relation between the previous and the new open sys-
tems. We prove that if the refinement relation holds, then the CFGs extracted from the available components in the original open system are sound over-approximations of the CFGs from the same components in the refined system. We say that a closed system implements an open system if they are in the refinement relation. Therefore, safety properties verified over the previously extracted CFGs hold for closed systems that implement the original open system.

Both CFG extraction algorithms have been implemented as the ConFlEx tool. It uses Sawja, an external library for the static analysis of Java bytecode, for the BIR transformation, and virtual method call resolution. We have tailored Sawja in several aspects: we have improved the analysis of explicit exceptions type, implement a sound virtual method call resolution algorithm for modular set-ups, implemented the data-structure to represent open programs, and the operations over it, such as the verification of the refinement relation. The ConFlEx tool implements, in addition to the extraction algorithms from BIR, the CFG representation data structure, and the caching of previous analysis, to enable incremental CFG extraction.

The experimental results confirm that the over-approximations, necessary for the second algorithm to be sound, have a significant impact on the size of the extracted control flow graphs. Also, the first algorithm is significantly more precise than the second one even for the analysis of closed systems, since it can use more precise virtual method call resolution algorithms. However, both algorithms have similar efficiency, and are liner w.r.t the number of instructions, and the fix-point computation of exception propagation is lightweight in practice. At last, the caching of previous analysis, and consequent incremental extraction of the newly arrived components, improve substantially the the analysis of open systems, when compared to an entire new extraction.

7.2 Future Work

The algorithms presented in the thesis focus on the soundness of the extracted models. Although we have successfully accomplished our goal, this has led to conservative design choices, and consequent excessive over-approximations. Thus, a natural improvement is the utilization of more precise analyses for virtual method call resolution, and specially for the computation of exceptional flow. For example, the BIR transformation inserts [nonnull] assertions whenever an object reference is accessed. Certainly not all instructions that operate with an object reference will in fact raise a NullPointerException. Also, the set of possible types for an exception raised explicitly is always estimated as its static type, plus all its sub-types. In practice, this set may be smaller. So, a possible improvement is to introduce points-to analysis [40][46] to filter out spurious exceptions.

The implementation of ConFlEx has room for several performance optimizations. Currently, the intra-procedural analysis extracts the CFGs from methods sequentially. So, a natural improvement would be to parallelize it. Ideally, each
thread would analyze one single method, and they would synchronize before proceeding to the inter-procedural phase. Another speed-up can be achieved by using a faster data structure for the caching of previous analysis. For example, the XML files currently in use produce very large files, which take a long time to be loaded and parsed. We can implement a more efficient caching of results by using some high-performance embedded database, such as Berkeley DB [45], or SQLite [30].

This work was an initial study about the generation of abstract software models for input to formal verification techniques. So far, we have generated CFGs which abstract from all data, and keep information about method invocations and exceptions. These models are suitable for the verification of a limited range of properties. Thus, we plan to study richer program models which are capable of capturing data, at the same time that we still target soundness, and pay attention to the state-space explosion. One promising direction is to combine Hoare logic verification with CFG extraction. Typically, Hoare logic verification generates assertions over a method’s control points when it proves that its pre- and post-conditions hold. Thus, we propose to add such assertions as atomic propositions in the abstract program model, so one can verify richer temporal safety properties.
Bibliography


[40] B. Steensgaard. Points-to analysis in almost linear time, 1996.


Appendix A

Weak Simulation

We present the notion of weak transition relation for models, which follows the standard definition from Milner [28]. As usual, we write \( p_i \xrightarrow{t} p_j \) to denote \( (p_i, t, p_j) \in \rightarrow \), for some relation \( \rightarrow \). Also, we use the \( \varepsilon \) label to denote silent transitions.

**Definition 15 (Weak transition relation).** Given an arbitrary model \( (S, L \cup \{ \varepsilon \}, \rightarrow, A, \lambda) \), the relations \( \Rightarrow \subseteq S \times S \), \( \beta \Rightarrow \subseteq S \times L \times S \) are defined as follows:

1. \( p_i \Rightarrow p_j \) means that there is a sequence of zero or more silent transitions from \( p_i \) to \( p_j \). Formally, \( \Rightarrow \overset{\text{def}}{=} \varepsilon \rightarrow^{*} \), the transitive reflexive closure of \( \varepsilon \rightarrow \).

2. \( p_i \xrightarrow{\beta} p_j \) means that there is a sequence containing a single visible transition labeled with \( \beta \), and zero or more silent transitions. Formally, \( \xrightarrow{\beta} \overset{\text{def}}{=} \Rightarrow \overset{\text{def}}{=} \Rightarrow \xrightarrow{\beta} \Rightarrow \)

Now we present the definition of weak simulation. Again, it is based on the standard notion, but instantiated over two method graphs for convenience.

**Definition 16 (Weak Simulation over Method Graphs).** Let \( \mathcal{M}_p = (S_p, L_p, \rightarrow_p, A_p, \lambda_p) : E_p \) and \( \mathcal{M}_q = (S_q, L_q, \rightarrow_q, A_q, \lambda_q) : E_q \) be two method graphs, and \( R \subseteq S_p \times S_q \). Then \( R \) is a weak simulation if for all \( (p_i, q_j) \in R \) the following holds:

1. \( \lambda_p(p_i) = \lambda_q(q_i) \)

2. if \( p_i \xrightarrow{\beta} p_j \) then there is \( q_j \in S_q \) such that \( q_i \xrightarrow{\beta} q_j \)

3. \( (p_j, q_j) \in R \).

We say that \( q \) (weakly) simulates \( p \) if \( (p, q) \in R \), for some weak simulation relation \( R \). Also we say that \( \mathcal{M}_q \) (weakly) simulates \( \mathcal{M}_p \) if for all \( p \in E_p \), there is \( q \in E_q \) such that \( q \) (weakly) simulates \( p \).
The following proposition is a consequence of Definition 16, also presented in the standard definition by Milner. To prove weak simulation, it suffices to show that for every edge produced by the direct algorithm (“strong” transition), there is a matching weak transition with the same label produced by the indirect algorithm.

**Proposition 2.** A relation $R$ is a weak simulation if and only if for all $(p_i, q_i) \in R$, the following holds:

1. if $p_i \xrightarrow{s} p_j$ then there is $q_j$ such that $q_i \xrightarrow{\varepsilon} q_j$ and $(p_j, q_j) \in R$.
2. if $p_i \xrightarrow{\beta} p_j$ then there is $q_j$ such that $q_i \xrightarrow{\beta} q_j$ and $(p_j, q_j) \in R$. 
Appendix B

Correctness of $bG \circ BC2BIR$

The $BC2BIR$ transformation may collapse several bytecode instructions into a single BIR instruction. Therefore, we divide bytecode instructions as *producer* instructions, i.e., those that produce at least one BIR instruction in function $BC2BIR_{\text{inst}}$, and *auxiliary* ones, i.e., those that produce none. This division can be deduced from Figure 2.12 (on page 23). For example, *store* and *invokevirtual* are producer instructions, while *add* and *push* are auxiliary.

We partition the bytecode instruction array into *bytecode segments*. These are subsequences delimited by producer instructions. Thus each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java bytecode Verifier (Def. 1). All methods in such a program must terminate with *return*, or *athrow*, which are producer instructions. Therefore, there can not be a set of contiguous instructions that is not delimited by a producer instruction.

Each bytecode segment is transformed into a set of contiguous instructions by $BC2BIR$. We call this set a *BIR segment*, which is a partition of the BIR instruction array. There exists a one-to-one mapping between bytecode segments and the BIR segments, which is also order-preserving. Thus, we can associate each instruction, either in the JBC or BIR arrays, to the unique index of its correspondent bytecode segment.

Figure 2.13 (on page 24) illustrates the partitioning of instructions into segments. Method *odd* has four bytecode (and BIR) segments, as indicated by the coloring. Producer instructions are underlined.

In the definition of the direct extraction algorithm in Figure 2.9 (on page 18), one can observe that all auxiliary instructions give rise to an internal transfer edge only. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path of internal transfer edges with the same size as the number of auxiliary instructions, followed by the edges generated for the producer instruction.

Below we illustrate the pattern for the path graph, being $i$ the position for the
first auxiliary instruction, and \(p\) the position of the producer instruction. In case
the number of auxiliary instructions is zero, then \(i = p\).

\[
\circ_m^1 \xrightarrow{x} \circ_m^{succ(1)} \xrightarrow{x} \circ_m^{succ(succ(1))} \xrightarrow{x} \ldots \circ_m^p \ldots
\]

Based on observations above, our main theorem states that the method graph
extracted using the indirect algorithm weakly simulates \((\text{cf. } \text{[28]})\) the method graph
using the direct algorithm. In the proof, we do not consider the abstract stacks,
since only the instructions are producer to produce the edges. Also, we use the
term 'simulates', instead of 'weakly simulates', for brevity.

**Theorem 1** (Structural Simulation of Method Graphs). Let \(\Gamma\) be a well-formed
Java bytecode program, and let \(\Gamma[m]\) be the implementation of method \(m\). Then
\((\text{bG} \circ \text{BC2BIR})(\Gamma[m])\) weakly simulates \(\text{mG}(\Gamma[m])\).

**Proof.** Let \(p\) range over indices in the bytecode instructions array, \(pc\) over indices
in the BIR instructions array, \(\circ_m^{p,x,y}\) over control nodes in \(\text{mG}(\Gamma[m])\), and \(\circ_m^{pc,x,y}\)
over control nodes in \((\text{bG} \circ \text{BC2BIR})(\Gamma[m])\). The control nodes are valuated with
two optional atomic propositions: \(x\), which is an exception type, and \(y\), which is
the atomic proposition \(r\) denoting a return point. Further, let \(\text{seg}_{JBC}(m, p)\) and
\(\text{seg}_{BIR}(m, pc)\) be two auxiliary functions that return the segment number that a
bytecode, or a BIR instruction belongs to, respectively, and let function \(\text{min}(s, x, y)\)
return the least index \(pc\) in the BIR segment \(s\) resulting in a node valuated with \(x\)
and \(y\).

We define a binary relation \(R\) as follows:

\[
R \overset{\text{def}}{=} \{ (\circ_m^{p,x,y}, \circ_m^{pc,x,y}) \mid \text{seg}_{JBC}(m, p) = \text{seg}_{BIR}(m, pc) \land pc = \text{min}(\text{seg}_{BIR}(m, pc), x, y) \}
\]

and show the relation to be a weak simulation in the standard fashion, following
Proposition \(\text{[2]}\) for every pair of nodes in \(R\), we match every strong transition from
the first node by a corresponding weak transition from the second node, so that
the target nodes are again related by \(R\). It is easy to establish that the entry nodes of
the sub-graphs produced by the two algorithms for the same bytecode segment are
related by \(R\), and hence the result.

The proof proceeds by case analysis on the type of the producer instruction of
the bytecode segment \(\text{seg}_{JBC}(m, p)\). The cases follow the subsets of JBC instructions
presented in Figure \(\text{[2.8]}\) which share the same extraction rule in the direct algorithm
for its instructions.

**Case** \(i \in \text{CmpInst}\)

There are two producer instructions in this subset: \texttt{nop} and \texttt{store}. We present
the case for \texttt{store} only, since it subdivides into two sub-cases, being the simplest
analogous to the case of \texttt{nop}. 

\[\text{\texttt{store}}\]
The direct extraction always produce a single transition from one normal node
to the node tagged with the successor of \( p \) in the instructions array:

\[
\mathfrak{m}G((p, \text{store}), H) = \{ \phi_m \xrightarrow{\circ} \phi_m^{\text{succ}(p)} \}
\]

The transformation BC2BIR\textsubscript{instr} can return either one or two assignments. Applying the extraction rules \( \mathfrak{b}G \) function, which have:

\[
\begin{align*}
\text{BC2BIR}_{\text{instr}}(p, \text{store}) &= \left\{ \begin{array}{ll}
[x:=a] & (\text{Case I}) \\
[t^0_{pc}:=x];[x:=a] & (\text{Case II})
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\mathfrak{b}G([x:=a]_{pc}, \tilde{H}) &= \{ \phi_m^{t_{pc}} \xrightarrow{\circ} \phi_m^{pc+1} \} & (\text{Case I}) \\
\mathfrak{b}G([t^0_{pc}:=x]_{pc}, \tilde{H}) &= \{ \phi_m^{t_{pc}} \xrightarrow{\circ} \phi_m^{pc+1} \} & (\text{Case II}) \\
\mathfrak{b}G([x:=a]_{pc+1}, \tilde{H}) &= \{ \phi_m^{pc+1} \xrightarrow{\circ} \phi_m^{pc+2} \}
\end{align*}
\]

In the Case I, where \text{store} produces a single assignment, we have that \((\phi_m^p, \phi_m^{pc}) \in \mathcal{R}\). The transition \( \phi_m^p \xrightarrow{\circ} \phi_m^{\text{succ}(p)} \), is matched by the weak transition \( \phi_m^{pc} \xrightarrow{\circ} \phi_m^{pc+1} \). Thus obviously also \((\phi_m^{\text{succ}(p)}, \phi_m^{pc+1}) \in \mathcal{R}\). The case for \text{nop} is analogous to this.

The Case II, where there are two assignments, also has \((\phi_m^p, \phi_m^{pc}) \in \mathcal{R}\). There transition \( \phi_m^p \xrightarrow{\circ} \phi_m^{\text{succ}(p)} \), is matched by the weak transition \( \phi_m^{pc} \xrightarrow{\circ} \phi_m^{pc+2} \), which transverses \( \phi_m^{pc+1} \). Then also \((\phi_m^{\text{succ}(p)}, \phi_m^{pc+2}) \in \mathcal{R}\).

**Case \( i \in \text{JmpInst} \)**

The only producer instruction in this subset is \texttt{goto} \( q \). The direct extraction produces a single transition from one normal node to the normal node tagged with the \( q \) position:

\[
\mathfrak{m}G((p, \text{goto } q), H) = \{ \phi_m^p \xrightarrow{\circ} \phi_m^q \}
\]

The transformation BC2BIR\textsubscript{instr} also returns a single instruction, which applied to \( \mathfrak{b}G \) function produces a single transition:

\[
\begin{align*}
\text{BC2BIR}_{\text{instr}}(p, \text{goto } q) &= \text{[goto } pc'] \\
\mathfrak{b}G([\text{goto } pc']_{pc'}, \tilde{H}) &= \{ \phi_m^{pc} \xrightarrow{\circ} \phi_m^{pc'} \}
\end{align*}
\]

We have that \((\phi_m^p, \phi_m^{pc}) \in \mathcal{R}\). Thee transition \( \phi_m^p \xrightarrow{\circ} \phi_m^q \) is matched by the corresponding transition \( \phi_m^{pc} \xrightarrow{\circ} \phi_m^{pc'} \). Thus \((\phi_m^q, \phi_m^{pc'}) \in \mathcal{R}\).

**Case \( i \in \text{CndInst} \)**

The only producer instruction in this subset is \texttt{if} \( q \). The direct extraction produces two transitions from the normal node tagged with position \( p \): one to successor control point, if the value on top of the operand stack is false; to the control point \( q \), otherwise:

\[
\mathfrak{m}G((p, \text{if } q), H) = \{ \phi_m^p \xrightarrow{\circ} \phi_m^{\text{succ}(p)}, \phi_m^p \xrightarrow{\circ} \phi_m^q \}
\]
The transformation \( \text{BC2BIR}_{\text{instr}} \) returns a single instruction, which applied to \( bG \) function produces two transitions:

\[
\text{BC2BIR}_{\text{instr}}(p, \text{if } q) = [\text{if } \text{expr } \text{pc}']
\]

\[
bG([\text{if } \text{expr } \text{pc}'], H) = \{ \circ_{\text{pc}} \overset{\circ_{\text{pc}+1}}{\rightarrow} \circ_{\text{pc}'} \}
\]

We have that \((\circ_{\text{pc}}, \circ_{\text{pc}'} \in R). The first transition \( \circ_{\text{pc}} \overset{\circ_{\text{succ}(p)}}{\rightarrow} \circ_{\text{pc}+1} \) is matched by \( \circ_{\text{pc}} \overset{\circ_{\text{pc}+1}}{\rightarrow} \circ_{\text{pc}'} \). Thus also \((\circ_{\text{pc}}, \circ_{\text{pc}'} \in R)\).

**Case** \( i \in \text{RetInst} \)

All return instructions are producer instruction. However, the direct algorithm does not produce transitions for them. It simply adds the atomic proposition \( r \) to the normal sink nodes tagged with the position of the return instruction.

Let \( q \) be the position of the return instruction. Also, let’s suppose w.l.o.g. that the direct algorithm has produced a transition from a node in position \( p \) to a node tagged with position \( q \), labeled with \( \lambda \).

The transformation \( \text{BC2BIR}_{\text{instr}} \) returns a single instruction, which applied to \( bG \) function produces a single transition:

\[
\text{BC2BIR}_{\text{instr}}(q, \text{return}) = [\text{return } \text{expr}]
\]

\[
bG([\text{return } \text{expr}], H) = \{ \circ_{\text{pc}} \overset{\circ_{\text{pc}+1}}{\rightarrow} \circ_{\text{pc}'} \}
\]

We have that \((\circ_{\text{pc}}, \circ_{\text{pc}'} \in R). The transition \( \circ_{\text{pc}} \overset{\circ_{\text{pc}+1}}{\rightarrow} \circ_{\text{pc}'} \) is matched by \( \circ_{\text{pc}} \overset{\circ_{\text{pc}+1}}{\rightarrow} \circ_{\text{pc}'} \). Thus also \((\circ_{\text{pc}}, \circ_{\text{pc}'} \in R)\).

**Case** \( i = \text{athrow} \)

Let \( X \) be the set containing the static type of the exception being thrown, and all of its sub-types. This set is the same for the direct and indirect extraction algorithms. Let \( x \in X \).

The direct extraction for the \( \text{athrow} \) instruction produces two edges, with the target node of the second edge depending on whether the exception \( x \) is caught within the same method it was raised or not:

\[
\text{mG}((p, \text{athrow}), H) = \begin{cases} 
\{ \circ_{p} \overset{\circ_{p}}{\rightarrow} \circ_{p,x} \}, & \text{if has handler} \\
\{ \circ_{p} \overset{\circ_{p}}{\rightarrow} \circ_{p,x} \}, & \text{otherwise}
\end{cases}
\]

The transformation \( \text{BC2BIR}_{\text{instr}} \) returns a single instruction. Then, similarly to \( \text{mG} \), the \( bG \) function produces two edges:

\[
\text{BC2BIR}_{\text{instr}}(p, \text{athrow}) = [\text{throw } x]
\]

\[
bG([\text{throw } x], H) = \begin{cases} 
\{ \circ_{p} \overset{\circ_{p}}{\rightarrow} \circ_{p,x} \}, & \text{if has handler} \\
\{ \circ_{p} \overset{\circ_{p}}{\rightarrow} \circ_{p,x} \}, & \text{otherwise}
\end{cases}
\]
We have that \((c^p_{m}, c^p_{pc}) \in R\). The transition \(c^p_{m} \xrightarrow{\epsilon} \bullet^p_{m} x\) is matched by the corresponding weak transition \(c^p_{pc} \rightarrow \bullet^p_{pc} x\). Thus obviously also \((\bullet^p_{m}, \bullet^p_{pc}) \in R\).

Next, there are two possibilities for the remaining transitions, depending on whether there is an exception handler for \(x\) in \(p\) and \(pc\). If there is a handler, then we get \(\bullet^p_{m} handle \rightarrow c^p_{q_{m}} \bullet^p_{pc} x handle \rightarrow c^p_{m} \), and clearly also \((c^p_{m}, c^p_{pc}) \in R\). If there is no exception handler for \(x\), we get \(\bullet^p_{m} handle \rightarrow \bullet^p_{m} x, r \bullet^p_{pc} handle \rightarrow \bullet^p_{pc} x, r\), and also \((\bullet^p_{m}, \bullet^p_{pc}) \in R\). This concludes the case.

**Case** \(i \in \text{XmpInst}\)

The instructions in this set follow to the next control point in case they terminate the execution normally, or can raise an exception if some condition was violated. We present this case for the \(\text{div}\) instruction, which can only raise the \(\text{XmpInst}\) (given the set \(RE\)). The case for other instructions in \(\text{XmpInst}\) is analogous.

The rule for the direct extraction produces one normal transition, for the case of successful execution. It also produces a pair of transitions for each exception instruction that the instruction can raise: one from a normal to an exceptional node; and the corresponding transition depending if there is an associated exception handler to the exception \(x\).

\[
\begin{align*}
\text{mG}(p, \text{div}), H) = \\
\begin{cases}
\{ c^p_{m} \xrightarrow{\epsilon} c^p_{\text{notzero}(p)}, c^p_{m} handle \rightarrow \bullet^p_{m} x, r \} & \text{if has handler} \\
\{ c^p_{m} \xrightarrow{\epsilon} c^p_{\text{notzero}(p)}, c^p_{m} handle \rightarrow \bullet^p_{m} x, r \} & \text{otherwise}
\end{cases}
\end{align*}
\]

The \(\text{BC2BIR}_{\text{instr}}\) transformation returns a single instruction, which is an assertion. The \(bG\) function produces three transitions: one to a normal node, denoting absence of exceptions, one to exceptional node, denoting the transfer of control to the JVM. The third transition varies if an exception handler is found. Thus we may have two sets of transitions:

\[
\begin{align*}
\text{BC2BIR}_{\text{instr}}(p, \text{div}) &= [\text{notzero}] \\
bG([\text{notzero}]_{pc}, H) = \\
\begin{cases}
\{ c^p_{pc} \xrightarrow{\epsilon} c^p_{pc+1}, c^p_{pc} handle \rightarrow \bullet^p_{pc} x, r \} & \text{if has handler} \\
\{ c^p_{pc} \xrightarrow{\epsilon} c^p_{pc+1}, c^p_{pc} handle \rightarrow \bullet^p_{pc} x, r \} & \text{otherwise}
\end{cases}
\end{align*}
\]

We have that \((c^p_{pc}, c^p_{pc+1}) \in R\). The transition \(c^p_{pc} \xrightarrow{\epsilon} c^p_{\text{notzero}(p)}\) is matched by \(c^p_{pc} \rightarrow c^p_{pc+1}\). Thus obviously also \((c^p_{\text{notzero}(p)}, c^p_{pc+1}) \in R\). There is a second transition from the same source node: \(c^p_{pc} handle \rightarrow \bullet^p_{pc} x, r\), which is matched by \(c^p_{pc} handle \rightarrow \bullet^p_{pc} x, r\). Thus, also \((\bullet^p_{pc}, \bullet^p_{pc}) \in R\).

There are two possibilities for the third transition, depending on whether there is an exception handler for \(x\) in \(p\) and \(pc\). If there is a handler, then we have \(\bullet^p_{m} \xrightarrow{\epsilon} c^p_{m}\).
APPENDIX B. CORRECTNESS OF $B\Gamma \circ BC2BIR$

and $\bullet_{m}^{pc,x} \Longrightarrow \circ_{m}^{pc'}$. Clearly $(\circ_{m}^{q}, \circ_{m}^{pc'}) \in R$. If there is no exception handler for $x$, we have $\bullet_{m}^{p,x} \xrightarrow{\text{handle}} \bullet_{m}^{p,x,r}$ and $\bullet_{m}^{p,x} \xrightarrow{\text{handle}} \bullet_{m}^{p,x,r}$. Thus, also $(\bullet_{m}^{p,x,r}, \bullet_{m}^{p,x,r}) \in R$.

Case $i \in \text{INVINST}$

This is the set of instructions which execute method invocations. We present cases for the instructions $	ext{invokespecial}$ and $	ext{invokevirtual}$. The case for $	ext{invokestatic}$ and $	ext{invokeinterface}$ are analogous to the former and the later, respectively. The remarkable difference between these pairs of instructions is that for the first there is only one possible receiver for the call; the later can have one-to-many receivers.

$i = \text{invokespecial}$

The $	ext{invokespecial}$ invokes methods that belong to the same class (including object constructors), or to the super class. The direct algorithm extracts a variable number of edges. It produces a minimum of three: one edge for the normal termination of the method, and two edges for the exceptional flow of $\rho = \text{NullPointerException}$ (control transfer to JVM and exception handling).

Also it produces pairs of edges for each exceptions propagated by the called method (denoted by $N_{p}^{i}$). We present the proof for a single exception $x$, propagated by the called method, and generalize to all the possible propagated exceptions.

The direct algorithm extracts the following set of edges:

$$mG((p, \text{invokespecial}), H) =$$

\[
\begin{cases}
\circ_{m} \xrightarrow{\text{call}(p)} \circ_{m}^{\text{null}(p)} \xrightarrow{\text{p} \xrightarrow{\text{null} \rightarrow \circ_{m}^{q}} \circ_{m}^{q}} \cup N_{p}^{i} & \text{if } \rho \text{ has handler} \\
\circ_{m} \xrightarrow{\text{call}(p)} \circ_{m}^{\text{null}(p)} \xrightarrow{\text{p} \xrightarrow{\text{null} \rightarrow \circ_{m}^{q}} \circ_{m}^{q}} \cup N_{p}^{i} & \text{otherwise}
\end{cases}
\]

The function $N_{p}^{i}$ produces the following pairs of edges for some exception $x$ propagated from a call to $C()$:

$$N_{p}^{i} = \begin{cases}
\{ \circ_{m}^{p} \xrightarrow{\text{handle}} \circ_{m}^{p,x}, \circ_{m}^{p,x} \xrightarrow{\text{call}} \circ_{m}^{q} \} & \text{If has handler} \\
\{ \circ_{m}^{p} \xrightarrow{\text{handle}} \circ_{m}^{p,x}, \circ_{m}^{p,x} \xrightarrow{\text{call}} \bullet_{m}^{p,x,r} \} & \text{otherwise}
\end{cases}$$

The $BC2BIR_{\text{instr}}$ transformation can return two different sets of instructions for the $	ext{invokespecial}$. First, we present the case for the invocation of object constructor. It returns a sequence of assignments to temporary variables ($[t_{pc}^{0} := x]$), denoted by $H_{\text{Save}}$; plus the call to $\text{new } C$:

$$BC2BIR_{\text{instr}}(p, \text{invokespecial}) = [H_{\text{Save}}(pc, as); t_{0} := \text{new } C(\ldots)]$$

Assignments to variables produce a single transition to the next control point. Thus, the extraction of $H_{\text{Save}}$ function produces a path graph:

$$bG(H_{\text{Save}}(pc, as), pc, H) = \{ \circ_{m}^{pc} \xrightarrow{\text{call}} \circ_{m}^{pc+1}, \circ_{m}^{pc+1} \xrightarrow{\text{call}} \circ_{m}^{pc+2}, \ldots, \circ_{m}^{pc'-1} \xrightarrow{\text{call}} \circ_{m}^{pc'} \}$$
The rule for the \texttt{[new C]} produces one normal edge for the case of successful execution, one pair of edges relative to the exceptional flow in case exception \(\varrho (=\text{NullPointerException})\), and a pair of edges for the propagation of exception \(x\) (denoted by \(\mathcal{N}^n_{pc}\)):

\[
\mathcal{B}_G([t_0^p_{pc}:=\text{new C}(...)_{pc'},{\bar{H}}]) = \begin{cases} 
\{ o_m^p \xrightarrow{C} o_m^p +1, o_m^p \xrightarrow{\varepsilon} \bullet^p_m, o_m^p \xrightarrow{\varepsilon} o_m^p \} \cup \mathcal{N}^c_{pc} & \text{If } \varrho \text{ has handler} \\
\{ o_m^p \xrightarrow{C} o_m^p +1, o_m^p \xrightarrow{\varepsilon} \bullet^p_m, o_m^p \xrightarrow{\varepsilon} o_m^p \} \cup \mathcal{N}^c_{pc} & \text{otherwise}
\end{cases}
\]

Also, function \(\mathcal{N}^n_{pc}\) can produce two different sets of edges, if there is or not a handler for exception \(x\).

\[
\mathcal{N}^n_{pc} = \begin{cases} 
\{ o_m^p \xrightarrow{\text{handle}} \bullet^p_m, o_m^p \xrightarrow{\varepsilon} \bullet^p_m \} & \text{If has handler} \\
\{ o_m^p \xrightarrow{\text{handle}} \bullet^p_m, o_m^p \xrightarrow{\varepsilon} \bullet^p_m \} & \text{otherwise}
\end{cases}
\]

We have that \((o^p_m, o^p_m) \in R\). The transition \( o^p_m \xrightarrow{C} o^p_m \xrightarrow{\text{succ}(p)}\), is matched by \(o^p_m \xrightarrow{\text{handle}} o^p_m\), which transverses all the nodes produces from \(H_{Save}\). Thus \((o^p_m \xrightarrow{\text{succ}(p)} o^p_m) \in R\). There is another transition from the same source node: \(o^p_m \xrightarrow{\text{handle}} o^p_m\). It is matched by \((o^p_m \xrightarrow{\text{handle}} o^p_m) \in R\) and \((o^p_m \xrightarrow{\text{handle}} o^p_m) \in R\). The next edge depends on the presence of a handler. If there is none, then the transition \(o^p_m \xrightarrow{\text{handle}} o^p_m\) is matched by \((o^p_m \xrightarrow{\text{handle}} o^p_m, o^p_m \xrightarrow{\varepsilon} o^p_m) \in R\). If there is a handler, then \((o^p_m \xrightarrow{\text{handle}} o^p_m) \in R\), and explanation is analogous. There is also the pair of transitions added by the propagation of exception \(x\), which is analogous: the first transition being into an exceptional node, and the second varies according to having a suitable handler for \(x\).

The second case for \texttt{invokespecial} is the one where the called method is not a constructor, but some method within the same class, or from the super class. The \texttt{BC2BIR_{instruct}} transformation returns the same instructions as before, but preceded by \texttt{[notnull]} instruction.

\[
\text{BC2BIR}_{\text{instruct}}(p, \text{invokespecial}) = \text{[notnull];[H_{Save}(pc,as);t_0:=\text{new C}(...)]]}
\]

Applying the extraction function to \texttt{[notnull]} we get the following edges, in addition to the same other edges as the previous case of \texttt{invokespecial}:

\[
\mathcal{B}_G([\text{notnull}]_{pc},{\bar{H}}) = \begin{cases} 
\{ o_m^p \xrightarrow{\varepsilon} o_m^p +1, o_m^p \xrightarrow{\text{handle}} \bullet^p_m, o_m^p \xrightarrow{\varepsilon} o_m^p \} & \text{If } \varrho \text{ has handler} \\
\{ o_m^p \xrightarrow{\varepsilon} o_m^p +1, o_m^p \xrightarrow{\text{handle}} \bullet^p_m, o_m^p \xrightarrow{\varepsilon} o_m^p \} & \text{otherwise}
\end{cases}
\]
Now we have that \((x_{m_n}, x_{m_m}) \in R\). The transition \(x_{m_n} \xrightarrow{n} x_{m_{\text{succ}(p)}}\), is matched by 
\(x_{m_m} \xrightarrow{n} x_{m_{\text{succ}(p)}}\), which transverses the node tagged with `notnull` position, and all the 
nodes produced from \(H\text{Save}\). Thus \((x_{m_{\text{succ}(p)}}, x_{m_m}) \in R\). There is a second transition 
from the same source node: \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\), and it is matched by 
\(x_{m_m} \xrightarrow{\text{notnull}} x_{m_m}\). Then, \((x_{m_n}, x_{m_m}) \in R\). Again, the next edge varies on the presence of a handler. If 
there is none, the transition \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\) is matched by \(x_{m_m} \xrightarrow{\text{notnull}} x_{m_m}\) and 
\((x_{m_n}, x_{m_m}) \in R\). If there is a handler, \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\) is matched by \(x_{m_m} \xrightarrow{\text{notnull}} x_{m_m}\). Therefore, \((x_{m_n}, x_{m_m}) \in R\).

Again, the explanation for transitions related to the propagation of exception \(x\) is analogous to the case for \(g\). There is a third transition from the same source 
node: \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\). It is matched by \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\), which transverses the node 
produced by `[notnull]`, and the nodes produced by \(H\text{Save}\). Thus, \((x_{m_n}, x_{m_m}) \in R\). If there is no handler for \(x\), the transition \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\) is matched by 
\(x_{m_m} \xrightarrow{\text{notnull}} x_{m_m}\), and \((x_{m_n}, x_{m_m}) \in R\). If there is a handler, \(x_{m_n} \xrightarrow{\text{notnull}} x_{m_m}\) is matched by \(x_{m_m} \xrightarrow{\text{notnull}} x_{m_m}\). Therefore, \((x_{m_n}, x_{m_m}) \in R\).

We now detail the case for `invokevirtual`, which invoke virtual methods. This 
case is similar to `invookespecial`, but the number of possible receivers to the 
method call may be more than one. We present the proof for a single method 
call receiver \(n\), and generalize it to all possible receivers.

The direct extraction extracts a variable number of edges. It produces a 
minimum of three: one edge for the normal execution of the method, and two edges for 
the exceptional flow of \(g = \text{NullPointerException}\) (control transfer to JVM and 
exception handling).

Also it may produce pairs of edges for exceptions propagated from methods 
called inside the current method (denoted by \(N^i_p\)). We state the proof for a single 
exception \(x\) by the called method, and generalize to all the possible propagated 
exceptions.

\[
mG((p, \text{invokevirtual}), H) =
\begin{cases}
\{ x_{m_n} \xrightarrow{n} x_{m_{\text{succ}(p)}}, x_{m_m} \xrightarrow{\text{notnull}} x_{m_{m_n}}, x_{m_m} \xrightarrow{\text{notnull}} x_{m_{m_m}} \} \cup N^i_p & \text{If } g \text{ has handler} \\
\{ x_{m_n} \xrightarrow{n} x_{m_{\text{succ}(p)}}, x_{m_m} \xrightarrow{\text{notnull}} x_{m_{m_n}}, x_{m_m} \xrightarrow{\text{notnull}} x_{m_{m_m}} \} \cup N^i_p & \text{otherwise}
\end{cases}
\]

The function \(N^i_p\) produces the following set of nodes for some exception \(x\) propa-
gated from a call to \(n()\):

\[
N^i_p = \begin{cases}
\{ x_{m_n} \xrightarrow{\text{notnull}} x_{m_{m_n}}, x_{m_m} \xrightarrow{\text{notnull}} x_{m_{m_m}} \} \cup N^i_p & \text{If has handler} \\
\{ x_{m_n} \xrightarrow{\text{notnull}} x_{m_{m_n}}, x_{m_m} \xrightarrow{\text{notnull}} x_{m_{m_m}} \} \cup N^i_p & \text{otherwise}
\end{cases}
\]

The \(\text{BC2BIR}_{\text{instr}}\) outputs a set of instructions, being the minimum two: the 
assertion `[notnull]` and the method invocation:
Applying the extraction function to \([\text{nonnull}]\) we get the following edges:

\[
bG([\text{nonnull}]_{\text{pc}}, \bar{H}) =
\begin{cases}
\{ \circ_m^p \xrightarrow{\varepsilon} \circ_m^{p+1}, \circ_m^p \xrightarrow{\text{handle}} \circ_m^0, \circ_m^p \xrightarrow{\varepsilon} \circ_m^* \} & \text{If } \rho \text{ has handler} \\
\{ \circ_m^p \xrightarrow{\varepsilon} \circ_m^{p+1}, \circ_m^p \xrightarrow{\text{handle}} \circ_m^0, \circ_m^p \xrightarrow{\varepsilon} \circ_m^* \} & \text{otherwise}
\end{cases}
\]

The assignment to temporary variable produces a single transition to the next control point. Thus, the extraction of \(HSave\) function produces a path graph:

\[
bG(HSave(\text{pc, as})_{\text{pc}}, \bar{H}) = \{ \circ_m^{pc+1} \xrightarrow{\varepsilon} \circ_m^{pc+2}, \ldots, \circ_m^{pc-1} \xrightarrow{\varepsilon} \circ_m^{pc} \}
\]

The rule for the \([\text{t}_n^p := \text{e.n}(\ldots)]\) produces one normal edge for the case of successful execution, and a pair of edges for the propagation of exception \(x\) (denoted by \(N_{\text{pc}}\)):

\[
bG([\text{t}_n^p := \text{e.n}(\ldots)]_{\text{pc}}, \bar{H}) = \{ \circ_m^{pc} \xrightarrow{n(\cdot)} \circ_m^{pc+1} \} \cup N_{\text{pc}}
\]

Also, function \(N_{\text{pc}}\) can produce two different sets of edges, if there is or not a handler for exception \(x\).

\[
N_{\text{pc}} = \begin{cases}
\{ \circ_m^{pc'} \xrightarrow{\text{handle}} \circ_m^p, \circ_m^{pc'} \xrightarrow{x} \circ_m^{pc',x} \} & \text{If handler} \\
\{ \circ_m^{pc'} \xrightarrow{\text{handle}} \circ_m^p, \circ_m^{pc'} \xrightarrow{x} \circ_m^{pc',x} \} & \text{otherwise}
\end{cases}
\]

The case for all nodes \(o_i^m\) in the path graph extracted from the auxiliary instructions is the same, and we have that \((o_i^m, o_i^p) \in R\), and the last transition in the path is \(o_i^m \xrightarrow{\varepsilon} o_i^p\).

Next we analyze the producer instruction. \((o_p^m, o_p^p) \in R\). There is the transition \(o_p^m \xrightarrow{n(\cdot)} o_m^{\text{suc}(p)}\), and there is also \(o_p^m \xrightarrow{\varepsilon} o_m^{pc^*}\), which transverses the node tagged with \(\text{nonnull}\) position, and all the nodes produces from \(HSave\). Thus \((o_m^{\text{suc}(p)}, o_m^{pc^*}) \in R\).

There is a second transition from the same source node: \(o_p^m \xrightarrow{\text{handle}} o_p^p\). There is also \(o_m^{pc} \xrightarrow{\varepsilon} o_m^*\), containing the edge produced by \([\text{nonnull}]\) and \((o_m^{pc}, o_m^p) \in R\). Again, the next edge varies on the presence of a handler or not. If there is none, then exists a transition \(o_m^p \xrightarrow{\text{handle}} o_p^0\). There is also \(o_m^{pc'} \xrightarrow{\varepsilon} o_m^{pc^*}\), and \((o_m^{pc'}, o_m^{pc^*}) \in R\). If there is a handler, then \(o_m^p \xrightarrow{\varepsilon} o_m^q\). Moreover, there is \(o_m^{pc'} \xrightarrow{\text{handle}} o_m^{pc'}\). Therefore, \((o_m^p, o_m^{pc^*}) \in R\).
Again, the explanation transitions of propagation of exception $x$ is analogous to the case for $y$. There is a third transition from the same source node: $\text{CS}_m \xrightarrow{\text{handle}} \text{CS}_m$. There is also $\text{CS}_m \xrightarrow{\text{handle}} \text{CS}_m$, which transvers the node produced by $[\text{notnull}]$, and the nodes produced by $H\text{Save.}$ and $([\text{notnull}], [\text{notnull}]) \in R$. If there is no handler for $x$, then exists a transition $\text{CS}_m \xrightarrow{\text{handle}} \text{CS}_m$. There is also $\text{CS}_m \xrightarrow{\text{handle}} \text{CS}_m$ and $([\text{notnull}], [\text{notnull}]) \in R$. If there is a handler, then $\text{CS}_m \xrightarrow{\text{handle}} \text{CS}_m$. Moreover, there is $\text{CS}_m \xrightarrow{\text{handle}} \text{CS}_m$. Therefore, $(\text{CS}_m, \text{CS}_m) \in R$. 

□