Modular Extraction of Control-Flow Graphs from Java Bytecode

ATTILIO PICOCO

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2012
Modular Extraction of Control-Flow Graphs from Java Bytecode

ATTILIO PICOCO

DD221X, Master’s Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 270 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Pedro de Carvalho Gomes
Examiner was Johan Håstad

TRITA-CSC-E 2012:091
ISRN-KTH/CSC/E--12/091--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.kth.se/csc
Abstract

Formal methods are a set of techniques proposed to achieve software quality and reliability. Among them, static formal verification checks whether a piece of software satisfies some property. Programs can have a large number of possible states, that is, execution configurations. Performing a static analysis on them may result an unfeasible task. Thus, formal verification techniques often use a model for the input program to overcome this problem. The control-flow graph (CFG) is one of the most common software models. In this work we are concerned about sound CFGs. Sound program models represent all the possible program states. They are used to verify safety properties, which state that something undesirable may not happen.

In the current project we present a method to extract control-flow graphs from Java Bytecode in a modular fashion. A modular approach allows to analyze incomplete systems, that is, software systems some of whose components are not available. Modularity is hard to achieve because of the inter-dependences that exist between the system components. Our solution is based on the notion of interfaces. This strategy requires the user to provide information about the missing components to resolve any of the inter-dependences involving them.

We base our approach on two previous results. Firstly, we extend a formal definition of the Java Virtual Machine framework to represent incomplete Bytecode programs. This allows us to manage the presence of different configurations of a program whenever any component is either newly provided, modified or added. We define a refinement relation as a set of formal constraints between two possible versions of the same Bytecode program. If such a relation exists following these formal rules, then the results about the verification of safety properties performed over the original version hold.

Next, we generalize a previous algorithm that performs the extraction of CFGs for closed programs. We adapt it to cope with those methods not provided. The algorithm uses the user-provided information to resolve inter-dependences between an available method and a missing one. Inter-dependences exist in terms of method invocations and propagation of exceptions along a sequence of method calls. Whenever there is a method invocation, we soundly over-approximate the possible receivers to the call, and the set of exceptions potentially propagated by them.

We implemented the modular extraction framework in the ConFlEx tool. The tool extracts CFGs from the avail-
able components using the interfaces provided by the user. Moreover, ConFlEx compares two versions of a program and checks if one refines the other, following the refinement rules we have defined. Finally, the tool can also store and load the results obtained, so they can be reused in a later execution.

Our technique performs substantial over-approximations to achieve soundness. Despite this, our test cases show that ConFlEx is efficient. Also, the extraction of the CFGs gets considerable speed-up by reusing results from previous analyses.
Acknowledgments

This Master’s Thesis is the conclusion of my path in the academic world during the last five years. I have worked on this project as part of my experience within the Erasmus programme at the Royal Institute of Technology (KTH) in Stockholm, Sweden, thanks to the opportunity that my home university (Politecnico of Turin) has offered to me. I then want to thank all the personnel in both institutions, being always helpful and patient with me as student.

I feel very obliged to Pedro de Carvalho Gomes, my supervisor at KTH in this project. He helped and drove me in all my steps while working on the project, always there to give me the best suggestions I could get for everything, cheering me up in the tough moments when I feel the end was hopelessly far from being near, and making me feel as part of a working team much better than I could expect.

I want to thank Prof. Dilian Gurov, the project supervisor in the background, always ready to discuss our choices and ideas, so that we could proceed in the development as better as possible.

I feel the need to thank Prof. Stefano Quer from Politecnico, who accepted the role of being my tutor in my home university without any problem at all. I thank him for always being patient and helpful, waiting for my news and updates, even for a long time.

I would have never faced up and enjoyed these five years-experience as much as I have done without knowing I had my family and friends always on my side.

First of all, I TRULY thank my parents. Without their deep support, this would have never been possible and become true. They have been always close to me, even if we have spent most of the time far away from each other. They have been there whenever I have needed it, with their precious advices and actions. I know that it will never be enough to thank them for all they have done. I know that this is the least I can do. Grazie di tutto, mamma e papa.

I want to thank all my relatives and family friends, who always believed in me and trusted in all the choices I have made and decisions I have taken.

I have to thank all my dear friends, the old and the new ones. Thanks to them, I have never felt alone, regardless of the time and place, of the moment and the distance: in Brindisi, in Turin, in Stockholm, always and everywhere. With them, I have had the time of my life whenever we spent the time together. The craziest and funniest moments ever.

These five years have been long, even if I feel they passed very quickly. I will remember them for ever, though. I THANK YOU ALL.
## Contents

1 Introduction ............................................. 1
   1.1 Organization ........................................ 4

2 Java Virtual Machine Framework .................. 5
   2.1 Java Bytecode Features .............................. 5
   2.2 Static And Dynamic Analysis ....................... 6
   2.3 Formal Java Virtual Machine ...................... 7

3 Program Models ........................................... 11

4 Extraction Algorithm For Closed Programs ...... 15
   4.1 The BIR Language .................................... 15
   4.2 Transformation From JBC To BIR .................. 17
   4.3 Extraction Of CFGs From BIR ...................... 18

5 Open Systems .............................................. 21
   5.1 Open Environment .................................... 21
   5.2 Refinement Rules .................................... 22
   5.3 Class Hierarchy and Exception Types .............. 24
   5.4 Modular Extraction .................................. 24
   5.5 Correctness .......................................... 26

6 Implementation ........................................... 27
   6.1 The Tool ............................................ 27
   6.2 The Module OpenEnv ................................ 29
   6.3 The Module MCA ...................................... 31
   6.4 Analysis Storage And Load ......................... 32
   6.5 Exception Handling .................................. 33

7 Practical Results ....................................... 35

8 Related Work ............................................ 39

9 Conclusion ................................................ 45
Chapter 1

Introduction

Over the last years the demand for software quality and reliability has increased. In order to achieve it, several techniques have been proposed. Among them, two approaches are often considered: testing methods and formal methods. The former aims to find as many actual bugs as possible in the code by running the software. The latter aims to prove certain software properties without executing the program, through the application of mathematical techniques. Unlike testing methods, formal methods result to be exhaustive, with respect to the domain of the property. The two approaches are complementary, and they usually detect different kinds of bugs.

Among formal methods, model checking is one of the techniques used to verify programs. In model checking the software is modeled into a graph representation. This technique explores exhaustively the state space of the model representing the software to verify if certain properties hold. Showing that a software model satisfies some given requirements is computationally costly. This happens because of the combinatorial explosion of the state space of the software. A state represents a snapshot of a specific execution of a program. The problem is characterized by the huge number of states regarding the given program. Thus, the exhaustive exploration of the entire state space results almost always an unfeasible task. The common approach to overcome this problem is to create abstract models of the program, to reduce the size of the state space.

One of the most common software models is the control-flow graph (CFG). A CFG represents the control-flow structure of the program. Specifically, the nodes represent the control points of the program, and the edges are the transitions that transfer the control from one control point to another. The presence of branches and join points in the program code determines different paths in the graph. Figure 1.1 shows an example of the control-flow graph extracted from a simple Java program. Nodes can be either normal (marked with $\circ$) or exceptional (marked with $\bullet$). In our notation, the edge labels can contain either a method name when the transition is referred to a method invocation, or handle to identify an exception handler block, or $\epsilon$ for any other type of instruction.

The structure of a CFG induces a behavior, that represents the sequence of
method calls in the activation stack. In this project we are interested in the extraction of sound CFGs that represent all the possible execution paths of a program. Such models are input to the verification of safety properties, which are the ones stating that something undesirable may not happen. Several formal analysis techniques use control-flow graphs as input software models. One example is a novel technique called compositional verification of temporal control-flow safety properties [13, 14]. This technique models the program with a CFG, which is checked against safety properties, expressed in some temporal logic. The technique has been implemented in a tool set called CVPP, designed to analyze Java Bytecode (JBC) programs. CVPP is encapsulated by the tool framework ProMoVer [25, 26, 27].

The current project focuses on the extraction of CFGs from Java Bytecode. The static analysis of JBC is hard because of its semantics. Some of its aspects heavily affect the control flow of the program. Specifically, the resolution of virtual method calls and the propagation of exceptions are major challenges. The latter especially has been neglected in similar studies because of the complexity it adds, or because it is shown that not dealing with them does not heavily impact the analysis accuracy [17, 30]. The two mentioned features impact the way software components relate. E.g., when analyzing a program statically, given a call of an object’s method, the possible object’s types have to be determined, to estimate the possible receivers for the method call. This requires the analysis of the class hierarchy. Another kind of inter-dependence between components is the return from an invoked method, caused when an exception is raised, but it is not caught. This leads the execution of the called method to terminate abruptly, and the exception is forwarded to the caller method.

Our goal is the extraction of models from open systems. A system is called open when the implementation of at least one of the components is missing. A modular strategy is the usual way to analyze open systems. Modularity is a concept that aims to treat each component of a system (e.g. a program) separately. Some of the reasons that motivate the research of a modular approach, in addition to the analysis of open systems, are the reuse of results from individual analyses, and the possibility to parallelize the analysis.

Figure 1.1: An example program and its Control-Flow Graph
In this project we define a modular extraction framework from the available Java Bytecode. Modularity is more difficult than the monolithic closed strategy because of the inter-dependence among the components. Whenever a component is not available, we are unaware of the way it relates with the other components. In our approach we use the notion of *interfaces* to represent the missing components. The interfaces contain the assumptions on what these components require and provide. Specifically, an interface contains the set of methods invoked and the set of exceptions handled, when the code will be provided. The use of interfaces allows us to construct the class hierarchy. Furthermore, we assume that a missing method can propagate any exception, except the handled exceptions declared in the interface. Such assumptions have to hold when a component becomes available. Our approach expects the user to provide this information.

We base our method on two previous works. Firstly, we generalize the formal definition of the Java Virtual Machine framework, as presented in [10], to represent Java Bytecode programs some of whose components are not available. Specifically, we extend the notion of *environment* to represent open systems. An open system is redefined whenever a component is provided. Thus, we define a *refinement* relation between two versions of a program. If the relation holds, following a set of formal rules, then the CFGs extracted previously represent an over-approximation of the current CFGs. Thus, previous results about the verification of safety properties still hold. Moreover, we may perform the extraction incrementally.

Secondly, we adapted the algorithm proposed in [1, 2] to perform the extraction of control-flow graphs from open systems. The original algorithm works on closed systems only. Our algorithm also manages inter-dependences to unavailable components by using the user-provided interfaces. The algorithm performs some over-approximations to achieve soundness. Specifically, we have to over-approximate the results obtained from the virtual method call resolution, as well as the exception propagation, wherever any missing component is involved. Any method invocation is resolved according only to the program class hierarchy, that is defined through the interfaces and is always available. We do not refer to the reachability of the code, as some components may be missing. The set of exceptions propagated is approximated to contain any possible exception defined for the program, excluding those exceptions declared in the interface. The user guarantees that these exceptions will be caught when the method will be provided eventually. That is, when the code will become available, if its contents match what the user has previously declared, the refinement relation will hold and any previous verification result would be preserved.

We expanded the software tool *ConFlEx* [7], already existing within the CVPP tool set, to handle open systems. This project brings a significant extension to the functionality of CVPP, as the tool set has been fully designed to support modularity. The tool is written in OCaml language. Its implementation relies on the *Sawja* library [3, 11]. We define the granularity of the software components to the level of methods. We use Java Annotations [21] to provide the interfaces of the missing methods. The extension of *ConFlEx* concerns to add the code necessary to manage
the representation of a program environment, to distinguish between components missing or provided, and to perform the over-approximations. We also implemented some Sawja modules for this purpose. The tool is adapted to extract the CFG from the available components by using the interfaces provided to handle the missing ones. Also, we have implemented the support for loading and storing partial results. This modular feature allows to reuse data obtained previously, improving the performance.

We validated our modular approach and the ConFlEx tool over several test-cases. Our experimental results show that the excessive over-approximation impacts the performance. This is inevitable though, as over-approximating is necessary for obtaining sound results. However, the tool compensates the approximations by reusing previous results, after providing any missing component. The reuse of results and the fact that the extraction is performed incrementally allow us to gain a relevant speed-up in the algorithm execution. Moreover, the final size of the CFG is related to the number of over-approximations performed during the algorithm. In some scenarios the CFG size decreases, because performing a refinement implies providing more information about the system inter-dependences. Thus, fewer over-approximations are needed. This is not always the case, as the size of the control-flow graph actually depends on the size of the program call graph. That is, the number of method invocations, and how many of them involve missing methods.

The current Master's Thesis has been developed as part of a wider project, presented in a related Technical Report [6]. The Thesis focuses on the practical aspects of the modular framework, that is implementation and practical results. The Technical Report focuses on the theoretical foundation of the modular extraction, and contains the correctness proofs of the developed technique.

1.1 Organization

The remainder of the Thesis is structured as follows. Chapter 2 contains the formal description of the Java Virtual Machine and Java Bytecode, referring to the work presented in [10]. Chapter 3 presents the program models we use to reason about Java programs. Chapter 4 describes the extraction algorithm for closed programs as presented in [1, 2]. Chapter 5 contains our definition of the open systems and the extraction algorithm adapted to handle missing components. Chapter 6 presents the implementation details of the tool. Chapter 7 shows the experimental results. Chapter 8 presents several related works existing in literature. In Chapter 9 we state our conclusions and suggest possible future improvements to the current work.
Chapter 2

Java Virtual Machine Framework

Our extraction algorithm takes Java Bytecode programs directly as input. Java Bytecode (JBC) is a target language which can be produced by compiling Java programs. The Java Virtual Machine (JVM) is the platform-independent framework that executes Bytecode. Thus, it is necessary to take into account all the features of Java Bytecode and the Java Virtual Machine mechanism relevant to us. In this chapter, we present the formal definitions and explanations needed to describe both the JBC and the JVM; most of them refer to the work by Freund and Mitchell [10].

2.1 Java Bytecode Features

Despite being a target language, Java Bytecode contains several aspects of an Object-Oriented programming (OOP) language: a JBC program is based on the definition of classes as the basic components. Classes are structures storing data and defining the means to work on them. A class contains fields and methods: the former are the variables that store the data; the latter are the procedures that access the data and handle them. Objects are entities associated to the definition of a class. Java Bytecode preserves several features of the OOP, such as inheritance, polymorphism, exceptions.

Inheritance is the mechanism that allows to reuse the definition of an existing class when defining a new one, with the possibility to extend it with its own fields and methods. Inheritance implies a behavior known as subtyping, that is, a class being defined shares some common properties with the inherited class. Furthermore, inheritance leads to the construction of a proper hierarchy. Every JBC program has its own class hierarchy, always having the same root (the class java.lang.Object), which any class descends from.

Polymorphism is the mechanism that allows a class to override methods. For a class, overriding a method means redefining it if already specified by one of its parent classes. Polymorphism allows several classes in a subhierarchy to have the same method declaration with a different implementation. When the method of an object is called and the object is instantiated as one of a given class, the method
whose implementation refers to this class would be invoked. A class can have as many children classes as desired: this implies different scenarios in terms of class definitions to consider.

Exceptions are entities used to signal if something goes wrong in the execution of the instruction flow. In JBC, exceptions are defined and treated as objects, all inheriting from the class \texttt{java.lang.Throwable}. Thus, the type of an exception is affected by the subtyping mechanism, in the same way ordinary objects are. They can be custom made or built in the language. Exceptions can be raised inside any called method and, if not caught by any calling method in the invocation sequence, they lead the program to terminate. Moreover, exceptions can often be raised at run-time for errors not foreseen when writing the code.

2.2 Static And Dynamic Analysis

During the execution of Java Bytecode, the control always belongs either to a method or to the Java Virtual Machine itself; it is transferred whenever a method is called, returns or an exception is raised. Static analysis methods aim to predict the possible control flow paths of a program. Thus, a static analysis tries to determine which are the reachable points for any program execution. Both subtyping and polymorphism come into action when any given code is executed. When a method is called or a field is accessed, the actual type of a referred object is determined dynamically. This means that analyzing statically the program does not allow to determine which is the exact class definition used.

The control-flow analysis has to take into account these aspects. It has to guarantee that the dynamic behavior of the program and thus all of its execution scenarios are considered properly. Performing a static analysis on a program assumes nothing but the information retrievable from the available source code of the program. In such a context, we necessarily have to over-approximate, since it is not possible to produce exact information. Nevertheless, we aim to get accurate results and thus to perform analyses that are as precise as possible. Practically, the most affected issues are the \textit{resolution of virtual method calls} and the \textit{exception handling}.

The resolution of virtual method calls is the action that sets the possible receiver types for a given method invocation. Then, the exact class type is determined to execute the instructions of the considered method. The static type of the receiver object is always known. However, other type definitions may be used, according to the execution flow of the program. Several scenarios can be considered: the object declaration, its instantiation, its use as an argument, whether a casting operation has been performed on it. The prediction of which is the exact class definition is not always possible, because of subtyping and polymorphism; it is determined at run-time only. Thus, a static analysis of the code has to take into account all the types considered for the caller object.

Whenever an exception is raised, the execution control passes to the Java Virtual Machine. Its task is to look for an entry into the exception table. An entry maps
2.3. FORMAL JAVA VIRTUAL MACHINE

the type of the exception raised and a range of program points into a code block handling the given exception. If an entry is found, then the exception is caught and the control is passed to the initial control point of the handling code; otherwise, the program execution is interrupted. In the exception table it is possible to find several handlers referring to the same exception or to its subtypes. Thus, different entries can be related to the same exception raised. Since exceptions are raised either within the code explicitly or at run-time for unexpected problems, it becomes hard to determine which exceptions might be raised while executing the program, and thus where the control flow moves.

2.3 Formal Java Virtual Machine

This section contains the formal definitions that model the JVM mechanism. We present those relevant to us, the way they are described by Freund and Mitchell in [10].

Each of the class files created by a Java Bytecode compiler contains the Bytecode instructions of the methods, the symbolic names and the type information about any data referred in the source code. The JVM uses this information to verify the code and to resolve references. References are used to identify all the components (classes, interfaces, methods, fields) from the Java language. They are generated from the grammar in Figure 2.1.

Figure 2.1: Grammar generating references

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

Figure 2.2 shows the Bytecode instruction set considered in our project. These instructions are those relevant in our extraction method of control-flow graphs from JBC. This set is a subset of the JVML instruction set described in [10].

All the declarations contained in each class file are modeled as an environment \( \Gamma \) (Figure 2.3). \( \Gamma \) is the union of three functions \( (\Gamma^I, \Gamma^C, \Gamma^M) \) that map each reference to its own definition, for all the interfaces, classes and methods, respectively. Given an interface, its definition contains the set of interfaces inherited and the set of methods provided. The contents of a class are the reference to its parent class, instance fields for the objects of that class, and the interfaces declared to be implemented by it. Finally, every method is defined by an array of instructions and a list of exception handlers.

Whenever a method of an object is called, we have to know which is the class definition used by that object, according to its previous use in the code. When the correct receiver method is determined, the flow control moves to it and its set of instructions is executed. The exact receiver for the call depends on the type of the
CHAPTER 2. JAVA VIRTUAL MACHINE FRAMEWORK

Class-Name ::= C | ...  
Field-Ref ::= f | ...  
Method-Ref ::= m | ...  

var\textsubscript{BC} ::= x | x_1 | x_2 | ... this  

instr\textsubscript{BC} ::= nop | push c | pop | dup | add | div  
| load var\textsubscript{BC} | store var\textsubscript{BC}  
| new C | constructor c  
| getfield f | putfield f  
| invokevirtual C.m  
| if pc | goto pc  
| vreturn | return

\textbf{Figure 2.2: Instructions of BC}

Γ\textsubscript{I} : Interface-Name \rightarrow \langle \text{interfaces} : \text{set of Interface-Name} \rangle  

Γ\textsubscript{C} : Class-Name \rightarrow \langle \text{super} : \text{Class-Name} \cup \{\text{None}\} \rangle  

Γ\textsubscript{M} : Method-Ref \rightarrow \langle \text{code} : \text{Instruction}^+ \rangle  

Γ = Γ\textsubscript{I} \cup Γ\textsubscript{C} \cup Γ\textsubscript{M}

\textbf{Figure 2.3: Environment Γ of a Java program}

object and is determined at run-time only. Thus, a static analysis aims to build a set of receivers that is as accurate as possible. Specifically, it takes into account the whole class subhierarchy with the object static type as root, so that all the possible subpaths of the control flow are considered correctly. Figure A.1 from \cite{10} shows the subtyping rules covering all the scenarios. They are used to check whether a class is a subtype of another in a formal way, within the environment Γ.
2.3. FORMAL JAVA VIRTUAL MACHINE

Any execution state of the Java Virtual Machine is defined as a configuration \( C = A ; h \). \( A \) is the stack of the activation calls and \( h \) represents a memory heap \([10]\). Figure 2.4 shows the definition of \( A \). The JVM behavior is defined as a non-deterministic transition system, with an infinite set of possible execution states and the transition rules, that is the semantics, to move from one configuration to another.

\[
\begin{align*}
A & ::= A' \mid \langle e \rangle_{exc} \cdot A' \\
A' & ::= \langle M, pc, f, s, z \rangle \cdot A' \mid \epsilon
\end{align*}
\]

Figure 2.4: Stack \( A \) of activation calls within a JVM configuration

Activation calls can either be \( \langle M, pc, f, s, z \rangle \) or \( \langle e \rangle_{exc} \). The former is composed by the method reference, the program counter, a map from variables to related values, the operand stack and information about the initialization of the object. The latter represents the case when exceptions are handled. In this case, the exception object being created is put on top of the stack.

In our project, not all the possible JBC programs are considered as possible input. Only well-formed programs are accepted, so that we can preserve the correctness of the analysis.

Definition 1 (Well-Formed Java Program). A well-formed Java Bytecode program is a closed program which passes the JVM Bytecode verification.

Passing the JVM Bytecode verification implies loading and executing a program successfully. The JBC verification consists of several tests, ranging from checking the correctness of the code format to asserting that pointers are not forged, that there are no memory access violations, that objects are accessed the way they are defined \([28]\).
Chapter 3

Program Models

In this chapter we present the models and definitions used to reason about Java programs. These definitions follow Amighi et al. [1].

Our goal is the definition of a model that represents the input program. First, we define the notion of model. A model is formally defined as follows:

**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 $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation assigning the set of atomic propositions that hold on each state $s \in S$. An initialized model is a pair $(\mathcal{M},E)$ with $\mathcal{M}$ a model and $E \subseteq S$ a set of entry states.

A model is a labeled transition system. It is a 4-tuple made of a set of states, a set of labels, a relation defining a transition from one state to another and a set of atomic propositions. Moreover, a mapping function is defined to assign a set of atomic propositions to states.

Control-flow graphs (CFG) are the abstractions we use to model a program. They are built by detailing the specification of the methods present in the program. This specification is defined as an instantiation of an initialized model that allows to get an abstract representation of sequential programs with procedures and exceptions. Here follows the definition of method specification.

**Definition 3 (Method Specification).** A Control-Flow Graph with exceptions for $m \in \text{Method-Ref}$ over sets $M \subseteq \text{Method-Ref}$ and $E \subseteq \text{EXCP}$ is a finite model $\mathcal{M}_m = (V_m,L_m,\rightarrow_m,A_m,\lambda_m)$ with $V_m$ the set of control nodes of $m$, $L_m = M \cup \{\epsilon,\text{handle}\}$ 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. $E_m \subseteq V_M$ is a non-empty set of entry control point(s) of $m$.

A method specification, thus a control-flow graph, is an instantiation of a model representing the control flow structure of a method. In it, nodes represent the
control points of the program and transitions represent the transfer of control from one point to the other. Nodes can be marked as the entry states when they match the control points where the method starts. An atomic proposition \( r \) marking a node \( v \in V_m \) indicates that the node represents a return point of the method. A node can be either normal (marked with a \( \circ \)) or exceptional (marked with a \( \bullet \)). The transition edges can be labeled either by a method name (marking a method call), by handle (marking a move to a handler code block), or by \( \epsilon \) (marking any other type of instruction). Method-Ref and EXCP are the infinite sets of all the possible methods and exceptions that may exist or be defined in Java, respectively.

In our modular approach, we use the notion of interfaces to specify the assumptions we make on what the components of the program require and provide. Thus, we can also reason about components that are not available, by using the information contained in their own interface. The interface is related to the control-flow graph representing the program or part of it (e.g., a method).

**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, \( E \subseteq \text{EXCP} \) 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.

An interface defines what are the methods provided and required, the exceptions that can be raised from them and the entry methods of the program, if any.

A method can be modeled with a control-flow graph. Thus, we now define the control-flow graph of a method. We say that the control-flow graph structure of a given method is defined by a pair of the method specification and the provided interface.

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

\[
\begin{align*}
- \quad \langle M_m, E_m \rangle : (\{m\}, I^-, E, M_e) & \text{ if } (M_m, E_m) \text{ is a method specification for } m \text{ over } I^-, E \text{ and } M_e, \\
- \quad G_1 \uplus G_2 : I_1 \cup I_2 & \text{ if } G_1 : I_1 \text{ and } G_2 : I_2.
\end{align*}
\]

Any method is mapped to a control-flow graph and to an interface. The CFG of the whole program is built from the disjoint union of all the CFG structures referring to the methods of the program.

After defining the structure, we then present the definition of the behavior of a CFG, induced by its structure. The CFG behavior models the JVM behavior as defined in [10]. The behavior is defined by an infinite state machine, since it represents the evolution of a stack containing the sequences of the activation records.
**Definition 6** (Control-Flow Graph Behavior). Let $G = (M, E) : I$ be a closed Control-Flow Graph with exceptions such that $M = (M, L, \rightarrow, A, \lambda)$. The behavior of $G$ is described by the specification $b(G)$, where $M_g = (S_g, L_g, \rightarrow_g, A_g, \lambda_g)$ such that:

- $S_g \in V \times (V)^*$, i.e., states are pairs of control nodes and stacks of control nodes,

- $L_g = \{\tau\} \cup L_g^C \cup L_g^X$ where $L_g^C = \{m_1| l \in \{\text{call, ret, xret}\}, m_1, m_2 \in I^+\}$ (the set of call and return labels) and $L_g^X = \{lx | l \in \{\text{throw, catch}\}, x \in \text{EXCP}\}$ (the set of exceptional transition labels).

- $A_g = A$ and $\lambda_g((v, \sigma)) = \lambda(v)$

- $\rightarrow_g \subseteq S_g \times S_g$ is the set of transitions in $\text{CFG}_m$ with the following rules:

  - $[\text{call}] (v_1, \sigma) \xrightarrow{m_1 \text{call} m_2}_g (v_2, v_1 \cdot \sigma)$ if $m_1, m_2 \in I^+$,
    
    $v_1 \xrightarrow{\text{call} m_2}_m v_1'$,
    
    $v_1' \in \text{next}(v_1)$, $v_1 \notin \text{EXCP}$
    
    $v_2 \models m_2$, $v_2 \in E$, $v_1 \models \neg r$

  - $[\text{return}] (v_2, v_1 \cdot \sigma) \xrightarrow{m_2 \text{return} m_1}_g (v_1', \sigma)$ if $m_1, m_2 \in I^+$,
    
    $v_2 \models m_2 \land r$, $v_1 \models m_1$
    
    $v_1' \notin \text{EXCP}$, $v_1' \in \text{next}(v_1)$

  - $[\text{xreturn}] (v_2, v_1 \cdot \sigma) \xrightarrow{m_2 \text{xreturn} m_1}_g (v_1', \sigma)$ if $m_1, m_2 \in I^+$, $v_2 \models m_2$,
    
    $v_1 \models m_1, v_2 \xrightarrow{\text{handle} m_2}_m v_2'$
    
    $v_1 \xrightarrow{\text{handle}} m_1 v_1'$,
    
    $v_2 \models x \land r$, $v_1 \not\models x$
    
    $v_1' \not\models x, x \in \text{EXCP}$

  - $[\text{transfer}] (v, \sigma) \xrightarrow{\tau}_g (v', \sigma)$ if $m \in I^+$, $v \xrightarrow{\epsilon}_m v'$, $v \models \neg r$
    
    $v \not\models \text{EXCP}$, $v' \not\models \text{EXCP}$

  - $[\text{throw}] (v, \sigma) \xrightarrow{\text{throw} x}_g (v', \sigma)$ if $m \in I^+$, $v \xrightarrow{\epsilon}_m v'$
    
    $v \models \neg r$, $v' \models \text{EXCP}$

  - $[\text{catch}] (v, \sigma) \xrightarrow{\text{catch} x}_g (v', \sigma)$ if $m \in I^+$, $v \xrightarrow{\text{handle} m}_m v'$
    
    $v \models \neg r \land \text{EXCP}$
    
    $v' \not\models \neg r$, $v' \not\models \text{EXCP}$

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 $\epsilon$, induce a control shift from one normal state to another, as defined by the $[\text{transfer}]$ rule. The $[\text{throw}]$ rule refers to an $\epsilon$ transition as well, but the next state node to visit is marked as exceptional. Both $[\text{catch}]$ and $[\text{xreturn}]$ rules are induced by a $\text{handle}$-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 of a raised exception, not caught in the returning method but potentially handled in the invoking method. The couple [call]-[return] refers to the case when a method is invoked by another method and the called returns to the calling one. In both [return] and [xreturn], the destination state in the returning method is marked as a return node.

We need to map the behavior of the control-flow graph to that of the Java Virtual Machine, so that we can perform the code analysis by using the CFGs. This relation is defined as follows:

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

\[
\begin{align*}
\theta(\langle m, p, f, s, z \rangle.A; h) &= \langle \phi^p_m, \theta(A; h) \rangle \\
\theta(\langle m, p, f, s, z \rangle.c; h) &= \langle \phi^p_m, c \rangle \\
\theta(\langle x \rangle_{\text{exc}}.\epsilon; h) &= \langle \bullet^x_m, x, r_m, \epsilon \rangle \\
\theta(\langle x \rangle_{\text{exc}}.\langle m, p, f, s, z \rangle.A; h) &= \langle \bullet^p_m, x, \theta(A; h) \rangle
\end{align*}
\]

Each record in the activation stack is defined by a program counter, current heap configuration, set of local variables, method signature and operand stack. Each state is a pair of a record and the current configuration of the stack. The function \( \theta \), defined recursively, maps a JVM configuration into a state of the CFG. The extracted structure of the CFG for a given method has to represent all the possible execution paths of the considered method. It is then possible to prove that the CFG behavior simulates that of the JVM [1].
Chapter 4

Extraction Algorithm For Closed Programs

Our modular CFG algorithm is based on a previous result from Amighi et al. [1, 2], which is designed for closed programs only. We generalize their algorithm to work with incomplete programs as well as fully provided programs. In our project we follow their proposal, that is, not to extract the control-flow graphs directly from the input JBC. Java Bytecode and the Java Virtual Machine mechanism are based on an intensive use of stacks and relative operations. This makes it difficult to perform any analysis on them. Thus, our algorithm performs the extraction of CFGs from Java Bytecode in two phases. The extraction is performed over an intermediate representation of Bytecode.

Both algorithms first transform the Java Bytecode into the Bytecode Intermediate Representation (BIR) [9]. Next, they extract the CFG from the BIR representation. The BIR language is implemented as a module contained in Sawja [3, 11], a library designed to perform static analysis on Java programs, which the implementation of our tool relies on. Here follows an overview of the BIR language and of the extraction algorithm for closed programs. For a deeper description, we refer to [9] for the former and to [1, 2] for the latter.

4.1 The BIR Language

The use of BIR language gives several advantages. First, the JVM is a stack-based machine. Thus, it requires some sort of stack analysis to determine the types of the operands. This type of analysis is not trivial, as it requires knowledge of the contents of the whole stack, while performing some operations on it. The transformation from JBC instructions into BIR generates a set of instructions that are no more stack-based. They are variable-based instead, and represent expression trees, differently from those of Java Bytecode. Next, the transformation provided generates code that usually has a smaller size than the original one. Finally, BIR also supports a subset of the Java unchecked exceptions [15]. It provides a set of instructions that
CHAPTER 4. EXTRACTION ALGORITHM FOR CLOSED PROGRAMS

perform assertions related to these exceptions.

In [9] the authors define the semantics of the BIR language. They prove that the transformation algorithm and the language semantics are correct, since they preserve the original semantics of the program, regarding the use of the relations over values, environments and observable events. Our extraction process is purely syntactic, so the correctness of the BIR semantics is unrelated to our work. However, it brings reliability to our correctness proof since the syntactic transformation from JBC into BIR is part of the proof itself.

Figure 4.1 shows the definitions for the syntax of the BIR language. BIR contains both local and temporary variables: the former are identifiers already defined in the Bytecode; the latter are new identifiers. It also provides expressions and instructions to handle variable and field assignments.

\[
\begin{align*}
\text{expr} & ::= c | \text{null} \quad \text{(constants)} \\
& | \text{expr} \oplus \text{expr} \quad \text{(arithmetic)} \\
& | \text{tvar} | \text{lvar} \quad \text{(variables)} \\
& | \text{expr} \cdot \text{f} \quad \text{(field access)}
\end{align*}
\]

\[
\begin{align*}
\text{lvar} & ::= \text{l} \mid \text{l}_1 \mid \text{l}_2 \ldots \quad \text{(local var.)}
\end{align*}
\]

\[
\begin{align*}
\text{tvar} & ::= \text{t} \mid \text{t}_1 \mid \text{t}_2 \ldots \quad \text{(temp. var.)}
\end{align*}
\]

\[
\begin{align*}
\text{target} & ::= \text{lvar} \\
& | \text{tvar} \\
& | \text{expr} \cdot \text{f}
\end{align*}
\]

\[
\begin{align*}
\text{Assignment} & ::= \text{target} := \text{expr} \\
\text{Return} & ::= \text{return} \text{expr} | \text{return} \\
\text{MethodCall} & ::= \text{expr}\cdot\text{ns}(\text{expr}, \ldots, \text{expr}) \\
& | \text{target} := \text{expr}\cdot\text{ns}(\text{expr}, \ldots, \text{expr}) \\
\text{NewObject} & ::= \text{target} := \text{new} \text{C}(\text{expr}, \ldots, \text{expr}) \\
\text{Assertion} & ::= \text{nonnull} \text{expr} | \text{notzero} \text{expr}
\end{align*}
\]

\[
\begin{align*}
\text{instr} & ::= \text{nop} | \text{if} \text{expr} \text{pc} | \text{goto} \text{pc} \\
& | \text{throw} \text{expr} | \text{mayinit} \text{C} \\
& | \text{Assignment} | \text{Return} \\
& | \text{MethodCall} | \text{NewObject} \\
& | \text{Assertion}
\end{align*}
\]

Figure 4.1: Expressions and Instructions of BIR

We must take into account the order of the object creation and of the exception throwing to define the transformation correctly. The two cases address the same problem: both orders have to be explicitly defined so they can hold, as done in the Bytecode. The former task is performed by the Java Virtual Machine in two separate steps: first, raw object allocation; then, constructor call. Only when the
object is created correctly, it can be referenced and used. In a sequence of object creations, the sequence order has to be maintained. Moreover, the steps related to different objects must not overlap. This is to preserve any dependence among the objects themselves. The latter case mentioned above implies that the transformation has to check dynamically for run-time errors due to different exceptions.

Let $C$ be a class in a Java program. BIR implements the two instructions $\texttt{[mayinit C]}$ and $\texttt{[var, := new C(e_1, \ldots, e_n)]}$ to handle the class initialization (the former), the allocation of the object and the call to its constructor (the latter). The class initialization is always performed before the others. This step occurs only once, that is, on the moment when a class is referenced for the first time, either for the creation of the object or for a static method call. The exception throwing order depends on the expression evaluation order. Specifically for the unchecked exceptions, BIR provides a solution based on the use of assertions on an expression $e$; if the check fails, a proper exception is raised. Some examples are $\texttt{[notzero e]}$, $\texttt{[nonnull e]}$. Figure 4.2 shows the unchecked exceptions supported by BIR.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\texttt{[nonnull]}$</td>
<td>$\texttt{NullPointerException}$</td>
</tr>
<tr>
<td>$\texttt{[checkbound]}$</td>
<td>$\texttt{IndexOutOfBoundsException}$</td>
</tr>
<tr>
<td>$\texttt{[notneg]}$</td>
<td>$\texttt{NegativeArraySizeException}$</td>
</tr>
<tr>
<td>$\texttt{[notzero]}$</td>
<td>$\texttt{ArithmeticException}$</td>
</tr>
<tr>
<td>$\texttt{[checkcast]}$</td>
<td>$\texttt{ClassCastException}$</td>
</tr>
<tr>
<td>$\texttt{[checkstore]}$</td>
<td>$\texttt{ArrayStoreException}$</td>
</tr>
</tbody>
</table>

Figure 4.2: Assertions provided by BIR, with the unchecked exceptions associated

### 4.2 Transformation From JBC To BIR

The algorithm transforms the input JBC code into a set of BIR instructions. The function $\texttt{BC2BIR}_{\text{instr}}$ is applied to each JBC instruction to perform the transformation. The transformation is defined as follows:

**Definition 8** (BIR Transformation Function). Let $\text{AbsStack} \in \text{Expr}^*$. The rules defining the instruction-wise transformation $\texttt{BC2BIR}_{\text{instr}} : \mathbb{N} \times \text{Instr}_{\text{JBC}} \times \text{AbsStack} \to ((\text{Instr}_{\text{BIR}})^* \times \text{AbsStack}) \cup \{\text{Fail}\}$ from Java Bytecode into BIR are given in Figure 4.3.

A key point of the algorithm is the way to manage the operand stack, and thus stack-based code. This is done by using a symbolic stack that allows a transformation from the original code to a set of 3-address instructions. Figure 4.3 shows the core of the algorithm, that is, the function mapping a Bytecode instruction into a list of BIR instructions. At the same time, these instructions are symbolically executed by using this abstract stack, which refers to symbolic expressions.
CHAPTER 4. EXTRACTION ALGORITHM FOR CLOSED PROGRAMS

<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 e₂]</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</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>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>store x</td>
<td>[x:=e] or [tᵢₚ :=x;x:=e]</td>
</tr>
<tr>
<td>putfield f</td>
<td>[nonnull e;FSave(pc,f,as);e.f:=e']</td>
</tr>
<tr>
<td>invokevirtual m</td>
<td>[nonnull e;HSave(pc,as);tᵢₚ :=e.m(e₁'...eₙ')]</td>
</tr>
<tr>
<td>invokespecial ns</td>
<td>[nonnull e;HSave(pc,as);tᵢₚ :=e.ns(e₁'...eₙ')] or</td>
</tr>
<tr>
<td></td>
<td>[HSave(pc,as);tᵢₚ :=new C(e₁'...eₙ)]]</td>
</tr>
</tbody>
</table>

Figure 4.3: BC2BIR<sub>instr</sub> - Transformation of a BC instruction at pc

Most of the instructions modify the abstract stack when they are symbolically executed. The transformation of return and jump instructions is simple, as well as that of [nop] instructions. The transformation of a subset of instructions, like [load x] and [push c], do not produce any BIR instruction, instead. The use of temporary variables (tᵢₚ) allows to handle those instructions affecting memory locations, such as [store x],[putfield f] and [invokevirtual C.m]. These variables store each element on the stack, whose value might change. Thus, temporary variables are necessary to preserve the consistency of the operand types. The instruction [new C] for the object creation is preceded by [mayinit C] for the class initialization. The reference to the new object is then pushed onto the stack.

The expressions representing the stack elements must not depend on the control flow. The control flow path is not linear when there are branches and join points. However, the size of the abstract stack has to remain the same, whereas the actual size of its content may vary during the transformation. The proposed solution is the definition of a normalized stack containing temporary variables that store the original stack elements.

4.3 Extraction Of CFGs From BIR

The algorithm is presented and described in [2]. Ours is developed by generalizing the original one to the case of code not provided yet. It is presented in Chapter 5. Here follows a brief overview of the original algorithm.

After the transformation of the Bytecode into a set of BIR instructions, the algorithm then performs the extraction of the control-flow graphs from them. Definition 9 presents the CFG extraction function.

Definition 9 (Control Flow Graph Extraction). The Control-Flow Graph extraction function bG : (Instr × N) × Eᵢ → P( V × Lₘ × V) is defined by the rules in Fig-
4.3. EXTRACTION OF CFGS FROM BIR

Given method \( m \) with \( ArInstr_m \) as its instruction array, the Control-Flow Graph for \( m \) is defined as \( bG(m) = \bigcup _{i_{pc} \in ArInstr_m} bG(i_{pc}, H_m) \), where \( i_{pc} \) denotes the instruction with array index \( pc \). Given a closed BIR program \( \Gamma_B \), its Control-Flow Graph is \( bG(\Gamma_B) = \bigcup _{m \in \Gamma_B} bG(m) \).

Let \( m \) be a method in a Java program. First, each Bytecode instruction in the body of \( m \) is mapped into a set of BIR instructions. Next, the whole set of BIR instructions of \( m \) is processed to produce a set of nodes and edges (that is, states and transitions in the control flow), by applying the extraction function to it. Thus, this set of nodes and edges results in a graph, representing the control flow of the whole method. Finally, the control flow of the program is represented by a control-flow graph that is the union of all the graphs that we have obtained for each method in the program. In Figure 4.4 we present the extraction rules for the algorithm.

\[
\mathcal{H}^pc_c = \begin{cases} 
\{ (pc_m, handle, \cdot_{m,pc}', \cdot_{m,pc}) \} & \text{if } h_H(pc, x) = pc' \neq 0 \\
\{ (pc_m, \cdot_{m,pc}) \} & \text{if } h_H(pc, x) = 0
\end{cases}
\]

\[
bG(i_{pc}, H) = \begin{cases} 
\{ (pc_m, \cdot_{m,pc}) \} & \text{if } i \in \text{Assignment} \cup \\
\{ \text{[nop]}, \text{[mayinit]} \} & \text{if } i \neq \text{[Assignment]} \cup \text{[mayinit]}
\end{cases}
\]

\[
N^pc_c = \bigcup _{n_{pc}' \in bG(n)} \{ (pc_m, handle, \cdot_{m,pc}) \} \cup \mathcal{H}^pc
\]

Figure 4.4: Extraction rules for Control-Flow Graphs from BIR

The simplest instructions are assignments, [nop] and [mayinit]. These produce a single edge from the current control node to the normal next one. Return instructions also add a single edge to a return node, that is a node referring to the same control point, but marked with the atomic proposition \( r \). Jumps can be either conditional (instruction [if expr pc']) or unconditional ([goto pc']): the former introduce two edges (to the next control point and to pc', respectively) to represent the branch; the latter add a single edge to the node referring to the control point pc'.

The [throw X] and method call instructions are treated similarly as they both depend on the static type of the object the instruction is invoked on (the exception thrown and the calling object, respectively). BIR provides the static type of the object only. For [throw X], let \( X \) be the set containing the static type and all of its
subtypes. For any \( x \in X \), an exceptional edge is added together with an appropriate handler edge, if any, according to the exception table. For normal method calls, let \( res^\alpha \) be the set of the method receivers determined after resolving the call. It will contain the method referring to the static class type of the original object and those referring to its children classes. In this case a normal edge will be added for each element \( n \in res^\alpha \).

Assertion instructions produce a branch: a normal edge if the exception is not raised, and an exceptional edge, together with a handler edge from the exception table, if any, when the assertion fails. The \([\text{new } C]\) instruction adds only one normal edge together with an exceptional edge because of a \( \text{NullPointer Exception} \).

The propagation of a given exception is performed backward according to the potential sequence of invocations. The function \( \mathcal{N}_pc \) adds exceptional edges about the considered exception to those program points where the caller method invokes the method propagating the exception. The function \( \mathcal{H}_pc \) checks if there is a suitable handler for the given exception. If there is, an edge is added to it. Otherwise the edge is added to an exceptional return node, and the propagation continues, until either a handler is found for it or there are no more methods in the sequence considered.

The implementation follows the strategy presented in [17]: the whole control flow containing both normal and exceptional data is extracted in two separate phases. The analysis is actually performed in two steps: first, an intra-procedural analysis; then an inter-procedural one. The former builds the graph for each method, according to the extraction rules. The latter manages the propagation of exceptions among the method calls.
Chapter 5

Open Systems

The goal of the current work is to perform the extraction of control-flow graphs from a Java Bytecode program in a modular way. This allows us to reason about particular configurations where at least one of the program components is missing. In this chapter we present the required definitions to accomplish this task. This Thesis focuses on the practical aspects of the modular approach presented. The theoretical concepts contained in this chapter follows those defined in [6].

5.1 Open Environment

In Chapter 2 we presented some definitions that are valid for a closed Java Bytecode program. A software system is closed if all of its components are available. However, there are scenarios in which some of the parts are missing, but it is still interesting to analyze them. Some example are systems depending on third-party software, or systems that are still under development.

We define a software system as open if it already specifies which are its components, but the implementation for at least one of them is not provided yet. Thus, a closed system is a special case of the open one. In order to represent and reason about open JBC systems, we first generalize the notion of the environment \( \Gamma \) (Figure 2.3) to them.

Figure 5.1 shows the definition of the open environment for Java Bytecode, represented by \( \Gamma_o \). Most of its contents are similar to the definition in Figure 2.3 for closed programs. \( \Gamma_o^I, \Gamma_o^C \) and \( \Gamma_o^M \) are the partial mappings from names to the program classes, interfaces and methods, respectively. \( \Gamma_o \) is defined as the union of \( \Gamma_o^I, \Gamma_o^C \) and \( \Gamma_o^M \). \( \Gamma_o^I \) and \( \Gamma_o^C \) are the same as their ‘closed’ counterparts \( \Gamma^I \) and \( \Gamma^C \).

Differences exist in the definition of \( \Gamma_o^M \). In an open environment we need to represent methods with empty bodies as well as concrete ones. This implies that the code array may contain no instruction at all. We comply with this requirement by defining the array as the closure of the Instruction set. Next, we change the semantics of the field \( \Gamma_o^M[m].\text{handlers} \) for any method \( m \in M \) not provided. In these cases it represents the set of exception types declared to be caught by the
missing method. That is, these exceptions will never be propagated when the
method code will be provided eventually.

5.2 Refinement Rules

A system is executable only when it becomes closed. Thus, we have to wait for all
the components still missing to be provided, to execute the program. Nevertheless,
the motivation for the modular approach is the possibility to infer properties that
hold for any closed system built incrementally from an open environment. When-
ever a component becomes available, we get a new configuration of the environment,
which may still be open. We define refinement as the action for which a component,
missing in the original system, gets provided into the new configuration, guarantee-
ing that the verification results on a previous environment still hold.

Let $\Gamma_o$ and $\Gamma'_o$ be two Java Bytecode open environments. We denote a refine-
ment as $\Gamma_o \preceq \Gamma'_o$ if $\Gamma_o$ refines $\Gamma'_o$. The relation $\preceq$ is defined in Figure 5.2. It
formally represents a set of constraints that we define to verify whether a refine-
ment is valid. Implementation is a special case of a refinement. We have it when a
closed environment refines an open one. Formally, $\Gamma$ implements $\Gamma_o$ if $\Gamma \preceq \Gamma_o$ and
$\Gamma^M[m].code \neq \emptyset$ for all methods $m$.

Before presenting the explanation of the refinement rules in Figure 5.2 we intro-
duce the auxiliary function \textit{EXCP}, which returns the set of exceptions propagated
by some method $\Gamma^M[m]$. An environment may be a refinement of another if each of
the partial mappings considered in both environments (that is, interfaces, classes,
methods) have the same domain. We need this constraint to assure that the en-
vironments are defined on the same set of components. A refinement between two
open environments holds if the environments of classes, interfaces, methods of the
former refine their corresponding in the latter.
5.2. REFINEMENT RULES

An interface $\Gamma^I_o[\omega].\text{interfaces} \subseteq \Gamma^I_o'[\omega].\text{interfaces}$ refines $\Gamma^I_o[\omega].\text{methods} = \Gamma^I_o'[\omega].\text{methods}$ if both provide the same set of methods, and if the new interface inherits a subset of the interfaces inherited by the old one. We need these requirements to guarantee that the virtual method call resolution produces a sound over-approximation of the possible call receivers. The interface environment $\Gamma^I_o$ refines $\Gamma^I_o'$ if they have the same domain, and a refinement relation exists between interfaces with the same name.

A class $\Gamma^C_o[\sigma].\text{super} = \Gamma^C_o'[\sigma].\text{super}$ refines $\Gamma^C_o[\sigma].\text{interfaces} \subseteq \Gamma^C_o'[\sigma].\text{interfaces}$ if it satisfies three conditions. First, the super class of the new class has to be the same as the old one’s parent class. Second, $\Gamma^M_o[\sigma].\text{code} \supseteq \Gamma^M_o'[\sigma].\text{code}$ and $\Gamma^M_o[\sigma].\text{handlers} \supseteq \Gamma^M_o'[\sigma].\text{handlers}$ if both are empty methods. Here we are relaxing the definitions. We are allowed to do that because if the original class implements more interfaces than the refined one, it means that the former is over-approximating the latter. Third, the set of fields defined in the refined class is a super-set of the same set in the original class. Intuitively, any analysis performed in the original class took into account all the fields defined by it. Any new field defined afterwards was not referenced in the original class. Thus, the presence of new fields does not compromise the validity of the previous analysis. Similarly to the case of the interfaces, the class environment $\Gamma^C_o$ refines $\Gamma^C_o'$ if their domains are the same, and a refinement holds between classes with the same name.

We define three different cases for the refinement between two methods $\Gamma^M_o[m]$ if both are concrete. The method is implemented if $\Gamma^M_o[\sigma].\text{code} \supseteq \Gamma^M_o'[\sigma].\text{code}$ and $\Gamma^M_o[\sigma].\text{handlers} = \Gamma^M_o'[\sigma].\text{handlers}$. If both methods are concrete, the method is implemented if $\Gamma^M_o[\sigma].\text{code} \supseteq \Gamma^M_o'[\sigma].\text{code}$ and $\Gamma^M_o[\sigma].\text{handlers} = \Gamma^M_o'[\sigma].\text{handlers}$. Finally, if both methods are empty methods, the method is implemented if $\Gamma^M_o[\sigma].\text{code} \supseteq \Gamma^M_o'[\sigma].\text{code}$ and $\Gamma^M_o[\sigma].\text{handlers} = \Gamma^M_o'[\sigma].\text{handlers}$.

Figure 5.2: Refinement Rules between open environments
and $\Gamma_o^M[m]$, according to the presence of code. The first is between two methods with empty bodies. In this case, $\Gamma_o^M[m]$ refines $\Gamma_o^M'[m]$ if the set of exceptions that the former guarantees not to propagate is a super-set of the exceptions not propagated by the original method. This implies that the refined method restricts more the constraint about the possible exceptions that may be propagated. The second is the refinement between two concrete methods, that is, their body is provided; both $\Gamma_o^M[m]$ and $\Gamma_o^M'[m]$ must have the same code and array. The third is the most relevant case, when a missing method $\Gamma_o^M'[m]$ is refined by a concrete method $\Gamma_o^M[m]$. The only constraint we define is that the refined method cannot propagate any of the exceptions declared to be caught and handled in the original method. Like the other two partial mappings, the method environment $\Gamma_o^M$ refines $\Gamma_o^M'$ if their domains contain the same elements, and there is a refinement relation between methods with the same name.

### 5.3 Class Hierarchy and Exception Types

Given the definition of an open environment, the refinement relation preserves the class hierarchy. That is, the subtype set for any class in the original model is always a super set of its subtype set in the refined model. This allows the virtual method call resolution to over-approximate the potential call receivers in a sound way. Definition 10 presents the class hierarchy for a Java program.

**Definition 10 (Type Hierarchy).** Let $\Gamma_o^I$ and $\Gamma_o^C$ be partial maps from interface and class names, respectively, to their attributes. The Java Type Hierarchy is defined as $\Gamma^C = \Gamma_o^C \cup \Gamma_o^I$.

As mentioned in Chapter 2, a Java exception is an object whose type is a subtype of the `java.lang.Throwable` class. Since the class hierarchy is preserved and is the same for a given open environment and all of its refinements, the set of exception types is known, and we can define it as follows.

**Definition 11 (Exception Types Set).** Let $\mathcal{L}$ be a given class hierarchy. We define $\mathcal{ANY} = \{\sigma \in \mathcal{L} | \sigma <: \text{java.lang.Throwable}\}$

The set of exception types, that we call $\mathcal{ANY}$, contains all the elements in the class hierarchy that are subtypes of the `java.lang.Throwable` class.

### 5.4 Modular Extraction

We aim to generalize the concepts valid for a closed environment, so they also hold for an open system. Thus, we need to define formally the CFG extraction rules for the provided methods, when an incomplete program is input to the analysis. These rules are based on those presented in Chapter 4, proposed for the extraction of CFGs from closed systems in [2].
Analysis is performed in the same way as for the case of closed programs. That is, the
side the method. The changes are noticed in the rules related to method calls:
The rules remain the same for those instructions transferring the flow control in-

Analyzing the exceptional flow in a modular scenario introduces more difficulties
than doing it for closed environments. The set of exceptions that a method raises
depends explicitly on the instructions from the method body. It is then impossible
to determine in advance the exact set of exception types that a missing method
may propagate. Besides throwing its own exceptions, a method might propagate
exceptions raised by other methods invoked in it.

In our modular approach we target a sound analysis. This means that we must
over-approximate the set of exceptions possibly propagated by a method whose code
is not available. At the same time we try to be restrictive as little as possible about
what the user has to provide. Thus, our proposal is to allow the user to annotate
exceptions raised by other methods invoked in it.

We define the Control-Flow Graph extraction function of a method \( \Gamma_o \) as
\( \Gamma_o : \text{METHOD} \times \text{ENV} \rightarrow (\mathcal{V} \times \mathcal{L} \times \mathcal{V})^* \). The rules are presented
in Figure 5.3.

\[ \begin{align*}
\mathcal{H}_{E}^C &= \left\{ \begin{array}{ll}
\{ \circ_m^x, \text{handle, } o_m^p \} & \text{if } h_M(p, x) = p' \\
\{ \circ_m^x, \text{handle, } o_m^p \} & \text{if } h_M(p, x) = 0 \\
\end{array} \right.
\]

\( \circ_G(i, H) = \left\{ \begin{array}{ll}
\{ (\circ_m^i, \varepsilon, o_m^i) \} & \text{if } i \in \text{Assignment} \cup \{
\text{[nop], [mayinit]} \} \\
\{ (\circ_m^i, \varepsilon, o_m^i), (o_m^i, \varepsilon, o_m^i) \} & \text{if } i = \text{[if } expr \ v \text{]} \\
\{ (\circ_m^i, \varepsilon, o_m^i) \} & \text{if } i = \text{[goto } v \text{]} \\
\{ (\circ_m^i, \varepsilon, o_m^i) \} & \text{if } i = \text{Return} \\
\bigcup_{x \in X} \{ (\circ_m^i, \varepsilon, o_m^i) \} & \text{if } i = \text{[throw } X \text{]} \\
\{ (\circ_m^i, \varepsilon, o_m^i), (o_m^i, \varepsilon, o_m^i) \} & \text{if } i = \text{Assertion} \\
\{ (\circ_m^i, \varepsilon, o_m^i), (o_m^i, \varepsilon, o_m^i) \} & \text{if } i = \text{NewObject} \\
\bigcup_{x \in E} \{ (\circ_m^i, n, o_m^i) \} & \text{if } i = \text{MethodCall} \\
\end{array} \right. \\
\end{align*} \]

\[ \begin{align*}
\mathcal{N}_E^C &= \left\{ \begin{array}{ll}
\bigcup_{x \in E} \{ (\circ_m^i, \text{handle, } o_m^p, x) \} & \text{if } |\Gamma_o[n]| \neq 0 \\
\bigcup_{x \in E} \{ (\circ_m^i, \text{handle, } o_m^p, x) \} & \text{otherwise} \\
\end{array} \right. \\
\end{align*} \]

Figure 5.3: Modular extraction rules for Control-Flow Graphs from BIR

The rules are similar to those shown in Figure 4.4, valid for closed programs. The
rules remain the same for those instructions transferring the flow control inside
the method. The changes are noticed in the rules related to method calls:
NewObject and MethodCall. Both refer to the function \( \mathcal{N} \), that extracts information
about the exceptional flow determined by the exception propagation, related to
the invocation of a method \( n \). Whenever the implementation of \( n \) is provided, the
analysis is performed in the same way as for the case of closed programs. That is,
analyzing its CFG, checking the exceptions it propagates, adding edges according to the possible presence of handlers. Whenever the body of the called method is not available, the only information we have is $\Gamma^M_o[m]$. It is retrievable using the function Catch. Given the set $\text{ANY}'$ (Definition 11), we consider the set of exceptions potentially propagated by $n$ as $\text{ANY}' - \text{Catch}$. Moreover, the Method-Call rule sets the algorithm used for the virtual call method resolution to Class Hierarchy Analysis (CHA) [8, 30]. The CHA algorithm is based on the analysis of the program class hierarchy, always known, and not on an analysis of the code reachability. Other algorithms depend on the presence of the code, so they cannot always be used correctly in a modular configuration.

5.5 Correctness

Several properties are preserved whenever we perform the modular extraction of control-flow graphs from a JBC program. The modular strategy is proved to be correct. For the proofs of correctness we refer to [6].

Whenever we refine an open environment with another one, the control-flow graph we extract from the refined environment is a subset of the CFG extracted for the original one. This means that the refinement relation follows a monotone progress, with respect to to the containment relation.

Theorem 1 (Containment of CFGs). Let $\Gamma_o$ and $\Gamma_o'$ be two Open Environments, and $m$ be some method signature such that $|\Gamma^M_o[m].\text{code}| > 0$ and $|\Gamma^M_o'[,m].\text{code}| > 0$. Then $\Gamma_o \preceq \Gamma_o' \implies oG(m, \Gamma_o) \subseteq oG(M(m, \Gamma_o')).$

Actually, this implies that the analysis performed over the original model is sound and it holds for any refinement.

We can extract CFGs from closed programs through our modular algorithm. As mentioned earlier, a closed system is a special case of an open one. Nevertheless, we aim to show the efficiency the algorithm and that the over-approximations are done only whenever it is required.

Proposition 1 (Modular Control-Flow Graph). Let $\Gamma_o$ be an open environment, where all its components are present. Then $bG(\Gamma_o[m]) = oG(\Gamma_o[m]).$

The CFG extracted by the modular algorithm $oG$ and the one extracted by $bG$ are the same for the same closed system, if the CHA algorithm is chosen as the virtual method call resolution mechanism.
Chapter 6

Implementation

This chapter contains a description of the practical aspects of the project. The implementation of the modular approach concerned several issues: the representation of an open environment, the way a component is either provided or marked as not available in the program, the resolution of virtual method calls and the handling of exceptions.

6.1 The Tool

The code for the modular extraction of CFGs has been implemented as an extension to the already existing tool ConFlEx [7]. It is written in OCaml language and it uses the Sawja library [3, 11]. It is executable from a command line as a stand-alone software.

At the beginning of the present work, the development stage of the tool was such that it could perform the extraction of CFGs for closed programs only. It could already take some arguments as program parameters, either directly from the command line or reading from an input file. The first improvement we did was to add code performing the validation of the input arguments. A module called OptionsProcessingFunctions has been developed and integrated to the tool. Its purpose is to process all the parameters and check their validity, according to their meaning. Specifically, it checks whether the execution configuration is valid, in terms of paths and program entries.

Other options have been added. Among them, the most relevant are those allowing the user to require the storage and the load of previous results, respectively (see Section 6.4). Moreover, we switched the semantics of those options specifying the type of analysis desired. The previous default behavior of the tool was to perform an intra-procedural analysis only. Now the default setting is to perform the inter-procedural analysis. We considered the inter-procedural analysis as likely to be performed more often than an intra-procedural one.

We aim to perform the analysis as accurately as possible. This implies to preserve the correct static type of the objects referenced along the code. E.g., this issue
affects the inter-procedural step, since we want to know precisely the type of the exceptions considered, either raised, propagated or caught. Thus, when the Bytecode is transformed into BIR, the generated expressions have to store the object types involved, if any. At the beginning of the project, this feature was lacking. Thus, we have added the code that accomplishes this task. Now any object (and so exceptions too) keeps its static type during the transformation process, instead of being generically considered as an object of the class java.lang.Object, which any other class derives from. As a consequence, this also applies to the associated expression.

Whenever the tool is executed, the program is parsed by the library functions. The parsed version of the program also includes the basic Java API components (classes and methods). We consider the API environment as a black box that we do not access explicitly, in terms of instructions. In fact, the analysis is performed only on the components of the client program. The API components are not considered, unless explicitly defined as part of the client program by the user. The client code might contain calls to API methods, though. Thus, we implemented some code to manage such occurrences. For any API method invocation, we assume that it might throw and propagate some exceptions, as we ignore what the API method does actually. Since we do not analyze its instructions, we take the throws declaration (if any) in the method signature as an assertion of what may be thrown inside that method. We then add some exceptional edges to the graph of the caller method related to these exceptions. This case is treated the same way any throw instruction is.

A module called DefCFG has also been developed. It is used to implement the control-flow graph entity. Its representation is separated from the tool itself, to leave them independent. The module contains the definition of the structures representing the nodes and edges of the graph, and all the functions necessary to create and access them. For now, the module has been integrated into our customized version of Sawja, aiming to make it a definitive part of it after revision.

We fixed some minor bugs present in the previous version of the software. Specifically, two were the most significant issues: one about the inter-procedural analysis and one about the creation of the call graph. The former problem concerned the correct update of the control-flow graph while performing the propagation. The algorithm went on using the CFG obtained from the intra-procedural analysis, instead of the current one that we get whenever new exceptional nodes and edges are added to it. The latter bug was about invoking inherited methods. The algorithm created a link between the calling method and the wrong called method, to update the call graph. Actually, the correct method that has to be considered is the one referring to the parent class actually implementing it. The reason is that the program methods are parsed according to which class provides a real implementation, whereas the Java Bytecode stores the call as if performed by a method from a child class (if any), when it is compiled.
6.2 The Module OpenEnv

Chapter 5 states that any program may be represented as an environment, that is open as long as at least one of its components has not been provided yet. A closed program is a special case of this general scenario, since all of its components are available.

First, we defined the way to represent missing components. We took into account several strategies. At first, we considered a class-based level of modularity. We evaluated the use of either abstract classes or interfaces to represent classes not provided. These techniques provided a clear, built-in way. However, we concluded that their use in the terms we mean could have created confusion in the semantics of the program itself, once the components would have become available. Moreover, the support for them provided by the library is quite poor, especially for the former solution.

Finally, the solution we have chosen is to use Java annotations [21]. This technique allows us to achieve more granularity in the level of modularity, shifting the range to method-based, and to have full support from the library in the implementation. Whenever the program is parsed, all the provided methods are transformed into a data structure that contains a field referencing to its annotations, among the others. Thus, given a method not available, the idea is still to provide it as a concrete one, but its body will be reduced to a single return instruction, according to the return type of its own signature. In this way we forge the method as not missing, but we use the annotation to mark it as actually not provided when its environment is defined. This marking allows us to filter it from the analysis and the actions performed by the tool, as if that method is currently not available, as expected.

The custom annotation we define is called GhostComponent (Figure 6.1). It contains two fields: req_meths and handlers.

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

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

Figure 6.1: The GhostComponent annotation used to mark a method as not available.
The field `req_meths` is an array of strings specifying those methods declared to be invoked at some point in the method code. These methods refer to the static type of that class actually implementing it. This allows us to make the representation of this information more compact. When the program is parsed, we create a list of classes from this set of required methods, by parsing the related string. Then, we add them and all of their children classes to the set of classes that have to be parsed. Thus, they are considered part of the domain and the virtual method call resolution can be performed correctly. When the environment for the program is being defined, the set of required methods will be built also including the signatures of those annotated but referring to the children classes, as the annotated methods are inherited by them. This does not apply for the constructor methods, as each class has its own. The format of each string has to be edited as a complete signature according to the use of packages, if any. The constructor methods have return type set to `void` and name to `<init>`. Figure 6.2 shows the generic format of the string (Figure 6.2a) and a usage example (Figure 6.2b).

\[
\text{RETURN_TYPE \hspace{3mm} PACKAGE.CLASS.METHOD(ARG_1, \ldots, ARG_N)}
\]

(a) Format of the string

\[
\text{java.lang.String \hspace{3mm} packA.classB.metC(int[], packD.classE)}
\]

(b) Usage example

Figure 6.2: Generic format of any string in the annotation field `req_meths` and an usage example.

The field `handlers` is an array of strings specifying the exceptions the user declares to be caught and handled by the method itself, when its code will be provided eventually. The name of the exception has to match with its namespace declaration (user-defined or not).

The user has to create a `.java` source file with the code in Figure 6.1 and put it into each directory containing an annotated method, respecting the use of packages, if any. That is, the annotation itself has to be part either of the package or of its subpaths. We use the standard built-in annotations `Retention` and `Target` to define the levels of visibility and granularity of the annotation. The former states that the annotation is only visible in the class file, but not at run-time; for our purposes this is enough. The latter determines that the elements which can be annotated are methods, both standard and constructor. It is not possible to annotate static class initializer blocks.

The module `OpenEnv` implements all the features that define the environment representing a program. The module contains the definition of the environment and the functions needed to work on it. The implementation follows the definition of Open Environment in Figure 5.1. The environment itself is in fact a structure containing three fields. Each field is a map associating a component of the program with its own environment. The components considered are interfaces, classes and...
6.3. THE MODULE MCA

methods. The environment is defined for the client program only. About method environments, we store a hash value for the code, instead of the whole instruction array, to have a compact representation.

The module provides several functions. One creates the environment from a program. During this step, any method is checked against our custom annotation to mark it as not provided, in the case. Another primary function checks if a method is marked as provided or not. Similarly, another function checks if a method has been provided in a later refinement of a program, given two environments (the old and the new one) representing it. Moreover, the module provides a function checking if a given environment is a refinement of another one. Its implementation follows directly the rules in Figure 5.2. Like the modules DefCFG and MCA, this module has also been inserted as part of Sawja.

6.3 The Module MCA

Given a set of instructions referencing an object, we know that the use of that object depends on the way it has been declared. That is, the object is bound to the static class type used in its declaration, either as a variable or as an argument. This is valid regardless of the class type which the object is an actual instantiation of. If parts of code are not available, we are generally unable to know all the possible scenarios which the object was created in. We cannot determine correctly the class definition related to the object, according to its creation. We may assume only that the definition of its static type will be used for that object.

In such a context, we have to guarantee that a call of an object’s method is resolved correctly. Practically, this implies to consider all the possible receiver types in the set built from the whole class type subhierarchy that has the static type of the object as root. Our focus is on the methods, but the same applies to the fields that the object can access.

In our modular approach we are interested in a virtual method call resolution mechanism that considers the class hierarchy rather than code reachability, as the user might not provide the whole code. Specifically, we need an implementation of the Class Hierarchy Analysis (CHA) strategy [8, 30]. The Sawja library provides Class Reachability Analysis (CRA) and Rapid Type Analysis (RTA) (plus RRTA - refinement of RTA) [3]. These techniques are refinements of CHA; they refine the set of receivers according to a reachability analysis of the whole code (i.e., all the actual instantiations are known). Moreover, they are originally defined for complete programs only, as well as CHA.

Nevertheless, a modular scenario is even more relevant with incomplete programs. Thus, we implemented a new module (MCA - Modular Class Analysis) that is a variant of CHA, also working with open systems. Practically, we adapted the code of CRA provided by Sawja for our purposes. The main change is the way the virtual method call resolution works. According to some tests we have done, CRA filters some of the possible receivers, depending on the fact that a class is
potentially reachable in the program. In our implementation, the set of receivers is over-approximated considering the static type of the object and all of its subtypes, following the definition of CHA in [30].

Another task of the module is to parse the user program, to perform the analysis. Since in a modular environment we do not know the contents of the methods not available, we basically do not know which classes are going to be referenced in them. We took in account two aspects to over-approximate the set of classes considered in the parsing process. This is necessary to define the domain of the program environment correctly, and thus to make the analysis sound. First, we included also those classes referenced in the signature of the methods. These classes refer to objects used as possible return values or arguments of the method. Second, we also considered all those classes whose definition is used if some objects are newly created or any method is invoked on them. We included them through the annotation GhostComponent (Figure 6.1). In fact, the use of these classes might be unpredictable if we only check the signature of the methods. Like the modules DefCFG and OpenEnv, this module has also been inserted in the custom version of the Sawja library.

### 6.4 Analysis Storage And Load

The goal of this project is to allow an user to develop a program and perform a static analysis on it by exploiting modularity as much as possible. Working incrementally implies for the user to know whether a certain version of the program is a correct refinement of a previous one. Thus, it is necessary to make old data available somehow. We accomplish this task by storing on request part of the results obtained by running the tool. Practically, we store the current environment (always defined for any program) together with the results at the end of the intra-procedural analysis step. Specifically, the control-flow graph, the list of exceptions explicitly thrown by each analyzed method, and the list of program points where a method is called and by which method (i.e. the call graph).

The storage of these data has two consequences: it allows to check if two versions of the program hold a refinement relation, and it speeds up the performance. The former is realized by comparing the two environments defined for the program versions considered, and checking whether one is a refinement of the other. The latter comes from reusing some data for all those components left unchanged while modifying the program. The underlying idea is that performing a new intra-procedural analysis on all the methods not modified will give the same results in terms of CFG, set of propagated exceptions and calling control points. This is valid even if a version is not a refinement of another, as long as the tool is executed on the program with the same configuration options. Whenever these data are loaded, they are updated so that any reference to components newly provided or changed is deleted. This is done to keep data coherent. These information will be completed by analyzing the methods not considered previously, for any new analysis performed.
6.5 Exception Handling

The storing step is performed after the intra-procedural analysis. In this phase of the algorithm all the methods are considered individually. This is not the case for the inter-procedural phase, since the results are affected by the sequence of method invocations in the program. The loading step is performed before the intra-procedural analysis, and it is optional, as well as the storage.

Practically, the tool creates three files, all with the same name: a text file, an XML file, a DTD file. The text file contains the options scenario under which the tool has been executed. This is required to check whether two versions of the program have been analyzed in a consistent way, when the user wants to load previous data. Thus, the loading step is actually performed only if the execution configurations of the two considered versions are the same. The XML file actually stores all the partial results listed previously, following the structure defined in the DTD file (Figure B.1). We store the DTD file so that the XML file containing the data can be validated during the loading step. We integrated the library \texttt{XML-Light} \[20\] to the tool, to parse both the XML and the DTD file and to perform the validation of the former against the latter.

The refinement check is performed as the last operation. If the user has required an intra-procedural analysis only, then the moment when the mentioned task is performed does not actually affect the results obtained, since previous data have been reused as much as possible during the execution in any case. If the user has chosen to perform an inter-procedural analysis, it is necessary to delay this task until the end of the analysis, instead. For all the methods that have been changed (newly provided or modified) from one version to the other, we have to check that the sets of handlers, declared as caught by the user in their own annotations, are still valid, according to those exceptions actually propagated by the methods, either explicitly thrown or raised from any method calls in them (as required by the rules in Figure 5.2).

6.5 Exception Handling

In the case that the body of a method is not provided, it is not possible to know in advance which types of exceptions can be raised whenever that method is called. Theoretically, any exception might be raised and possibly propagated to the caller method. If the user requires to perform an inter-procedural analysis, it is necessary to determine which exceptions can be propagated by a method not provided, whenever this is invoked. In the implementation, we assume that this set of exceptions, called $\mathcal{ANY}$ (Chapter 5), is built from the union of three sets:

1. exceptions generated from the assertion instructions possibly inserted in the transformation (Figure 4.2);
2. exceptions defined by the user in the program;
3. exceptions declared as potentially throwable by the API methods.

33
Then, this set is filtered with the set of exceptions that the user declares as caught and handled directly by the considered method, when its code will be provided eventually.
Chapter 7

Practical Results

We have tested our modular extraction algorithm on three different software: JFlex version 1.4.3 [18]; JavaCUP version 11a beta [12]; Jasmin version 2.4 [19]. All the tests have been done on a machine with an Intel i3 2.27 GHz processor and 4GB of RAM. The data collected are: number of JBC and BIR instructions; number of nodes and edges of the CFG after the inter-procedural analysis; time needed to perform the intra-procedural analysis, the inter-procedural one, to parse the program, and to load previous results, if any. We have considered six different scenarios.

1. Open Environment: 4 classes explicitly not provided - Virtual Method Call Resolution: MCA

2.  
   a) Open Environment: 1 class explicitly not provided - Virtual Method Call Resolution: MCA - Load of previous results
   
   b) Open Environment: 1 class explicitly not provided - Virtual Method Call Resolution: MCA - No load of previous results

3.  
   a) Closed Environment: all classes provided - Virtual Method Call Resolution: MCA - Load of previous results
   
   b) Closed Environment: all classes provided - Virtual Method Call Resolution: MCA - No load of previous results

4. Closed Environment: all classes provided - Virtual Method Call Resolution: CRA

The scenarios reflect the incremental refinements we have performed on each software. Specifically, by performing the first refinement over the initial open system (step 1), we have scenarios 2a and 2b. Then, by implementing the system into a closed Environment, we get scenarios 3a, 3b, and 4.

The missing classes are: NFA, SemCheck, RegExp and IntCharSet for JFlex; symbol_set, terminal_set, lalr_state and production for JavaCUP; ClassFile,
InsnInfo, Scanner and parser for Jasmin. The classes not provided yet in steps 2a and 2b are: NFA for JFlex; production for JavaCUP; Scanner for Jasmin. By ‘missing classes’ we mean that all the methods of these classes have been annotated, and thus considered as not provided. The extraction concerned only the available classes. Classes from API are not extracted. Static class initializer blocks are not taken into account, as it is not possible to annotate them at all. The results we have collected are resumed in Table 7.1. The time is given in seconds.

The number of instructions (both JBC and BIR) increases whenever a new component is fully provided. This happens because those methods not provided are not considered into the counting. As expected, the number of BIR instructions is much smaller than that of JBC instructions. Roughly, it is a bit more than one third.

The number of nodes and edges (that is, the size of the control-flow graph) does not always decrease after a refinement. This can be seen in passing from step 1 to either step 2a or step 2b for the tests on JavaCUP. A refinement generally implies providing more information about the inter-dependences among the components, and thus fewer over-approximations are performed. However, this value varies because it actually depends on the size of the call graph, and specifically on how many invocations to not provided methods exist in the code. A call to a not provided method implies to propagate a set of exceptions equal to \( ANY \), but those declared to be handled in the future code. When providing a new method, a smaller set of exceptions might be explicitly thrown, but the propagation phase might consider new sets of exceptions if the code of the given method also contains calls to methods not available yet.

As expected, the time needed to perform the intra-procedural analysis decreases considerably whenever we reuse data from previous executions, instead of computing a brand new one for each element. The reason is that the actual number of methods for which we extract the CFG is much smaller, since at any refinement step those considered for the extraction are only those freshly provided.

A strange behavior has been detected about the performance of the inter-procedural analysis. We expect that the times are comparable either we load old data or we compute everything again from scratch, whenever this task is performed. In fact, the storing and loading steps affect the intra-procedural analysis only. As it can be seen from the statistics, it takes much longer to perform the inter-procedural analysis if we load something than doing it without loading. The results are coherent though, since we get the same CFG size in both cases. We could not figure out the reason despite analyzing the code and the tests carefully. Further investigation is needed.

The statistics show that the loading time is relevant, as well as the parsing time. This means that the loading time cannot be overlooked when considering the total execution time.

We have also executed the tool using the CRA mechanism once we have provided the complete program. The data are coherent: the number of instructions and the CFG size are the same using CRA and MCA. Since CRA is a refinement of MCA,
we get smaller times to parse the program and to compute the intra- and the inter-procedural analysis.
Chapter 8

Related Work

In this chapter we review works related to the definition of modularity and the way to approach it, the use of graphs as models, the Java Bytecode type-system, the mechanism of virtual method calls resolution and exception handling, and the use of API libraries.

The method we developed for the modular analysis of Java Bytecode is based on the formal definition of the Java Virtual Machine (JVM) framework, as it is presented by Freund and Mitchell [10]. The authors present a type system for Java Bytecode and propose a formal description of its semantics. Moreover, the authors model a complete Java Bytecode program through the notion of environment. The work by Freund and Mitchell applies to closed programs only. Our approach extends it to open systems. We propose our definition of open environment by generalizing their definitions regarding closed environments. Furthermore, the algorithm presented in this work reuses the type system proposed in [10] to approximate the control flow soundly. Specifically, we reuse the subtyping relation for the virtual method call resolution and the exception propagation. Chapter 2 contains a deeper explanation about the work by Freund and Mitchell and the concepts we reuse for our own approach.

Java Bytecode is a machine language; nevertheless, it has several features of Object-Oriented languages, such as class inheritance and polymorphism. Thus, it shares similar challenges with them, with respect to static analysis of code: the resolution of virtual method calls and the exception handling. Both issues affect the program control flow.

Whenever a method is invoked, the static analysis of a program has to determine which are the methods that may receive the call. Several algorithms about the resolution of virtual method calls exist in literature. The most popular are Class Hierarchy Analysis (CHA) [8, 30] and Rapid Type Analysis (RTA) [29, 30]. 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 instantiations 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 and their refinements are presented as valid for closed programs only. Nevertheless, in a modular scenario a virtual method invocation must be resolved despite the absence of some components. Such a context implies that any algorithm based on code analysis might not give correct results. We have to over-approximate the receivers soundly by using the class hierarchy only. Thus, we use a variant of CHA also working with open systems, as presented in Section 6.3.

Sundaresan et al. [29] 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 project 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 [30] 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 [30]: 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.

Another major issue is exceptional flows. Exceptions are usually neglected [17, 30], due to the complexity they induce, and because not treating them has usually small consequences on the accuracy of the analyses. However, in literature some approaches exist about the construction of the exceptional flows. To the best of our knowledge, they are presented as valid for scenarios taking into account closed systems only. Our work mainly differs from them since it proposes a solution suitable to the analysis of open systems within a modular set-up. Moreover, the presented approach is proved to be correct [6]. Thus, it turns out to be a novel result with respect to modular analysis of exceptional flows.

Sinha et al. [23, 24] present an algorithm that extracts control-flow graphs from
both Java source code and Bytecode. The extraction is done in two steps. First, it performs an intra-procedural analysis. Next, an inter-procedural analysis. The former determines the exceptional return nodes for the exceptions not handled. The latter performs the exception propagation. This algorithm considers explicit exceptions only. Our work is based on a similar division of tasks. However, it mainly differs from this about the exceptions considered. The use of BIR [9] allows us to consider also a subset of unchecked exceptions by inserting some assertions in the transformation. Moreover, during the Bytecode analysis, we can explicitly retrieve the static type of an exception raised because of the BIR transformation, whereas the authors do not discuss the way this task is accomplished.

Jiang et al. [16] extend the work of Sinha et al. to C++ source code. The exception mechanisms of C++ and Java share similarities: the same try-catch scheme, as well as the exception propagation. C++ lacks of finally handler blocks and unchecked exceptions, though. The authors do not consider the types of the exceptions. Thus, they over-approximate the control flows by building connections between any explicit throw instruction within a try block to all its possible catch blocks, and considering a potential exceptional termination for any called method containing a throw. Our work considers the static types of exceptions, instead. This allows our algorithm to extract control-flow graphs that are more accurate, as well as to provide information about the exceptions considered.

Choi et al. [4] propose an algorithm that performs the extraction of CFGs with exceptional flows by using an intermediate representation for the input code. They suggest to use a stackless representation, as well as explicit assertions that check those control points that may potentially raise exceptions. Demange et al. followed this approach in [9] when defining the BIR language. Our work is very similar to this. The main difference is that we define the extraction rule scheme formally and the extraction algorithm is proved to be correct in [6].

Jo and Chang [17] present a method to construct control-flow graphs from Java programs. This technique decouples the construction of the normal flow from that of the exceptional flow, instead of computing them at the same time. Through a study of frequency on some benchmark software, it is actually shown that the mutual dependence rarely happens, thus the effects of this approximation are usually negligible. In our work we treat also exceptions to over-approximate the program behavior soundly. Thus, we consider this work as a good starting point for our own approach, because the technique described takes into consideration all the possible cases where an exception can be raised. Moreover, we follow their strategy to compute the two flow graphs separately, first performing an intra-procedural analysis, and then an inter-procedural one. Also, this solution may benefit the performance and allow the reuse of previous results. The main difference between this work and ours is the possibility to determine the static type of the exceptions and to consider unchecked exceptions. Both aspects are missing in the former. We consider them by using the BIR transformation.

Amighi et al. [1, 2] present the implementation of an algorithm that extracts control-flow graphs from closed Java Bytecode programs. It addresses both the
virtual method call resolution and the exception propagation issues. For a deeper description of the algorithm, we refer to Chapter 4. The authors prove it to be formally correct: the extracted graph models the control flows of the given program, soundly over-approximating its behavior. The structure of the control-flow graph induces a behavior that simulates the Java Virtual Machine behavior. The authors achieve such a result by proving the structural simulation over the algorithm mentioned. In [13] they prove that structural simulation implies behavioral simulation. This project is a further development of this work, as it aims to adapt it by defining an algorithm that extract control-flow graphs from Bytecode programs modularly. Thus, our algorithm can be also executed over open systems.

The goal of this project is to propose a modular approach to analyze software systems. Modularity is hard to achieve because of the inter-dependences existing among the system components. Cousot [5] presents four strategies to perform static analyses of programs in a modular fashion. The first is the simplification-based: in it, the inter-dependences are simplified globally; this saves resources but is not efficient. The second is the worst-case, where no information whatsoever about inter-dependences 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-dependences; 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 [22] is a concrete example of a program analysis strategy that is conceived from a modular point of view. Specifically, the author presents a technique to identify correctly side-effect-free methods in Java software, that is, methods whose invocation does not provoke state changes observable by the executed code. The approach is based on a class analysis, that determines the possible classes which an object reference points to. The result obtained is a call graph defined by points-to pairs of a field/variable/method and the instance of a class, and call edges between methods. Methods with indirect side effects are detected through backward propagation on the call graph. This technique is designed and developed to work for incomplete programs. Thus, it targets open systems as well as our work. Its goal is to detect those methods being free from side-effects. In our work, we assume instead that all the methods in the API libraries are side-effect-free. However, even though the described approach differs from ours, it might be used to prove or disprove such assumption. We could apply this technique preliminarily, before the extraction and analysis. Possible side effects might be added to the control-flow graph extracted. This idea can be considered as a future refinement of our project. The benefit is to get more accurate results, since it is very usual to call a large number of API functions within a Java program. Thus, the mentioned assumption may result weak.
To conclude, the approach we developed follows one meaningful guideline. The result we get is the extraction of control-flow graphs that are tailored for formal verification, especially for compositional verification. The algorithm is formally proved to be correct in [6]. Also, we define the way to extract the CFG of the whole program from a compositional point of view. A concrete example of suitable usage of our method is CVPP [13, 14], and the related tool set ProMoVer [25, 26, 27]. Gurov et al. [13, 14] describe the technique of the compositional verification, to prove the correctness of a system by analyzing its components and verifying local properties on them. Their result is the definition of a program model, that is, a control-flow graph on which structural properties are verified. These structural properties simulate behavioral properties of the program. The compositional aspect comes into action when the global control-flow graph is built as disjoint union of the local control-flow graphs, obtained for each considered component. In [25, 26] they present the tool framework ProMoVer [27], which provides and implements tools and interfaces to perform the compositional verification of temporal safety properties of compound applications, through several and logically separated steps. Their work is an example of formal verification technique that requires modularity. Our project results very suitable for the purposes of this work. The modular extraction algorithm can be applied to the system components, to get the control-flow graphs for each of them. This does not prevent the analysis of the program to start when some components are not available, as the partial results can be stored and reused when necessary. Moreover, the extraction tool itself can become part of the framework ProMoVer, fully designed to support modularity.
In the present work we presented a modular framework for the extraction of sound control-flow graphs from Java Bytecode open systems. The method we developed allows us to analyze programs in which the implementation of some of its components is missing. The result is correct as long as proper information about the components not available are provided. Moreover, whenever any missing component is newly provided and a new analysis is required after, our approach exploits the results obtained in previous executions. Practically, we reuse the control-flow graphs of those components which have not been changed, and the data related to them (call graph and set of explicitly propagated exceptions). In fact, they remain the same at the end of the intra-procedural step, before performing the inter-procedural analysis, since they are not affected by a missing component that has been freshly made available.

We based our work on two previous results. First, we generalized the formal definition of the Java Virtual Machine framework in [10] and specifically the notion of environment, valid for closed programs only, for representing open systems formally. We also defined a refinement relation between two open environments representing two different versions of the same program. The relation is specified as a set of formal rules. We check whether the relation holds, to state that properties and results inferred and obtained previously are still valid for the refined version of the program.

Moreover, we adapted the algorithm presented in [2] to perform the extraction of the CFGs modularly. We extended it to comply with the case where missing components are involved. Any call to a method not available has consequences in both the intra-procedural and the inter-procedural analysis. For the former, it implies to over-approximate the resolved set of receivers to the subhierarchy having as root the static type of the object whose the method is invoked. For the latter, the set of exceptions possibly propagated has to be over-approximated to a set made of any potential exception of the program, but those declared to be handled by the method code, when it will be provided eventually.

The framework is implemented in the ConFlEx tool [7]. The code concerns
mainly two issues. Firstly, the definition of the environment for the current program and all of its features, e.g. checking whether a program has been refined by providing a new element. Secondly, the over-approximations needed for the virtual method call resolution and the exception handling in a modular scenario. This also implies the definition of a sound class hierarchy, as we are unaware of the code potentially implemented in a method. Finally, we implemented the code for storing and loading the results of any analysis performed.

The experimental statistics show that the over-approximations impact the performance, although they are necessary to achieve sound results. However, the tool compensates this effect by reusing previous results when providing any missing components and re-executing the tool. The intra-procedural step actually gains a meaningful speed up when loading the related results obtained in a previous execution. About the time needed to perform the inter-procedural analysis, we detected a strange behaviour. Specifically, the time needed for the mentioned phase varies significantly, according to whether we load previous results. On the contrary, we expected it not to be affected by the loading step. This situation demands further investigation.

9.1 Future Work

The proposed method perform excessive over-approximations, necessary to achieve soundness. As it is, the approach we presented provides possibilities for further developments and refinements. From a theoretical point of view, future efforts might be put in refining some of the concerned issues, e.g. the definition of the set of exceptions possibly propagated by a method whose instructions are not available. The idea is to work on these issues to be restrictive as little as possible when defining the refinement rules for an open environment, and at the same time to over-approximate as little as possible when computing some specific set of data.

About the implementation, some other solutions can be proposed for tuning performance, storing and presenting the analysis results in the output. A possible technique to speed up the tool is to parallelize the extraction step of the algorithm when executing the intra-procedural step, by using a multithreading strategy. Each thread would analyze either one single method or all the methods of a single class, as the number of methods considered and related threads created in the former suggestion might be too large. The partial results related to each class would be merged at the end of the intra-procedural analysis, before proceeding with the inter-procedural phase. A different solution about the storage and loading of data may be to use DataBase structures instead of text files. In this way the time needed to perform the storing and the loading steps could improve significantly.
Appendix A

Subtyping In Java Bytecode

Figure A.1 shows the subtyping rules as presented in [10]. They are used to check whether a class is a subtype of another formally, within the environment $\Gamma$ that is defined for a closed JBC program, as described in Chapter 2.
### APPENDIX A. SUBTYPING IN JAVA BYTECODE

<table>
<thead>
<tr>
<th>$\langle :$C REF $\rangle$</th>
<th>$\langle :$C SUPER $\rangle$</th>
<th>$\langle :$I REF $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma \in \text{Class-Name}$</td>
<td>$\Gamma \vdash \sigma \langle :$C $\sigma \rangle$</td>
<td>$\omega \in \text{Interface-Name}$</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma &lt; :$C $\sigma$</td>
<td>$\Gamma \vdash \sigma_1 &lt; :$C $\sigma_2$</td>
<td>$\Gamma \vdash \omega &lt; :$I $\omega$</td>
</tr>
<tr>
<td>$\Gamma[\sigma_2].\text{super} = \sigma_3$</td>
<td>$\Gamma[\sigma_2].\text{super} = \sigma_3$</td>
<td>$\Gamma \vdash \sigma_1 &lt; :$C $\sigma_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$I SUPER $\rangle$</th>
<th>$\langle :$A PRIM $\rangle$</th>
<th>$\langle :$A REF $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \omega_1 &lt; :$I $\omega_2$</td>
<td>$\Gamma \vdash \tau \in \text{Prim}$</td>
<td>$\Gamma \vdash \tau_1 &lt; :$A $\tau_2$</td>
</tr>
<tr>
<td>$\omega_2 \in \Gamma[\omega_3].\text{interfaces}$</td>
<td>$\Gamma \vdash \tau &lt; :$A $\tau$</td>
<td>$\Gamma \vdash \tau_1 &lt; :$A $\tau_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$R CLASS $\rangle$</th>
<th>$\langle :$R INTERFACE $\rangle$</th>
<th>$\langle :$R CLASS INT $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \sigma_1 &lt; :$C $\sigma_2$</td>
<td>$\Gamma \vdash \sigma_1 &lt; :$R $\sigma_2$</td>
<td>$\omega_1 \in \Gamma[\omega_3].\text{interfaces}$</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma_1 &lt; :$R $\sigma_2$</td>
<td>$\Gamma \vdash \omega_1 &lt; :$I $\omega_2$</td>
<td>$\Gamma \vdash \omega_1 &lt; :$I $\omega_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma_1 &lt; :$R $\sigma_2$</td>
<td>$\Gamma \vdash \omega_1 &lt; :$R $\omega_2$</td>
<td>$\Gamma \vdash \sigma_1 &lt; :$R $\omega_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$R Null $\rangle$</th>
<th>$\langle :$R ARRAY ARRAY $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau \in \text{Simple-Ref} \cup \text{Array} \cup {\text{Null}}$</td>
<td>$\Gamma \vdash \text{Null} &lt; :$R $\tau$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau &lt; :$R $\tau'$</td>
<td>$\Gamma \vdash \text{Array} \tau &lt; :$R $\text{Array} \tau'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$R INT Object $\rangle$</th>
<th>$\langle :$R ARRAY Object $\rangle$</th>
<th>$\langle :$ REF $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega \in \text{Interface-Name}$</td>
<td>$(\text{Array} \tau) \in \text{Array}$</td>
<td>$\Gamma \vdash \tau_1 &lt; :$R $\tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \omega &lt; :$R Object</td>
<td>$\Gamma \vdash (\text{Array} \tau) &lt; :$R Object</td>
<td>$\Gamma \vdash \tau_1 &lt; :$R $\tau_2$</td>
</tr>
<tr>
<td>$\langle :$ REF $\rangle$</td>
<td>$\langle :$ Top $\rangle$</td>
<td>$\langle :$ $\epsilon$ $\rangle$</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>$\tau \in \text{Uninit} \cup \text{Prim} \cup \text{Ret}$</td>
<td>$\tau \in \text{Ref} \cup \text{Prim} \cup \text{Ret} \cup {\text{Top}}$</td>
<td>$\langle :$ Top $\rangle$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau &lt; : \tau$</td>
<td>$\Gamma \vdash \tau &lt; : \text{Top}$</td>
<td>$\langle :$ $\epsilon$ $\rangle$</td>
</tr>
<tr>
<td>$\langle :$ MAP $\rangle$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$ SEQ $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \alpha_1 &lt; : \alpha_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 &lt; : \tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \cdot \alpha_1 &lt; : \tau_2 \cdot \alpha_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$ LOOP $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \alpha_1 &lt; : \alpha_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 &lt; : \tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \cdot \alpha_1 &lt; : \tau_2 \cdot \alpha_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$ IF $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \alpha_1 &lt; : \alpha_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 &lt; : \tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \cdot \alpha_1 &lt; : \tau_2 \cdot \alpha_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$ami $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \alpha_1 &lt; : \alpha_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 &lt; : \tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \cdot \alpha_1 &lt; : \tau_2 \cdot \alpha_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$ NULL $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \alpha_1 &lt; : \alpha_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 &lt; : \tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \cdot \alpha_1 &lt; : \tau_2 \cdot \alpha_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\langle :$ SEQ $\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \alpha_1 &lt; : \alpha_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 &lt; : \tau_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \cdot \alpha_1 &lt; : \tau_2 \cdot \alpha_2$</td>
</tr>
</tbody>
</table>

$\sigma \in \text{Class-Name}$ $\Gamma \vdash \sigma \langle :$C $\sigma \rangle$

$\langle :$I INTERFACE $\rangle$ $\langle :$A PRIM $\rangle$ $\langle :$A REF $\rangle$

$\langle :$R CLASS INT $\rangle$

$\langle :$R Null $\rangle$ $\langle :$R ARRAY ARRAY $\rangle$

$\langle :$R INT Object $\rangle$ $\langle :$R ARRAY Object $\rangle$ $\langle :$ REF $\rangle$

$\langle :$ REF $\rangle$ $\langle :$ Top $\rangle$ $\langle :$ $\epsilon$ $\rangle$

$\langle :$ SEQ $\rangle$ $\langle :$ MAP $\rangle$

$\langle :$ LOOP $\rangle$

$\langle :$ami $\rangle$

$\langle :$ NULL $\rangle$

$\langle :$ SEQ $\rangle$

$\langle :$ NULL $\rangle$

Figure A.1: Subtyping rules.
Appendix B

Open Environment Storing File
Definition

The open environment representing a given program is stored in an XML file, as described in Chapter 6. The XML file also stores some other partial results obtained during an execution of the tool. The data are stored following the structure defined in a separate DTD file, shown in Figure B.1. The content of the XML file is validated against the DTD format during the loading step.
APPENDIX B. OPEN ENVIRONMENT STORING FILE DEFINITION

<!ELEMENT open_env_cfg_pp_map_all_excp_prop (open_env , cfg , pp_map , all_excp_prop)>
<!ELEMENT open_env (interface_section , class_section , method_section)>
<!ELEMENT interface_section (interface_elem ∗)>
<!ELEMENT interface_elem (int_cl_name , interface_list , method_list)>
<!ELEMENT int_cl_name (#PCDATA)>
<!ELEMENT interface_list (interface ∗)>
<!ELEMENT interface (int_cl_name)>
<!ELEMENT method_list (methods)>
<!ELEMENT methods (method ∗)>
<!ELEMENT method (int_cl_name , method_signature)>
<!ELEMENT method_signature (ret , met_name , arg_list)>
<!ELEMENT ret (#PCDATA)>
<!ELEMENT met_name (#PCDATA)>
<!ELEMENT arg_list (arg ∗)>
<!ELEMENT arg (#PCDATA)>
<!ELEMENT class_section (class_elem ∗)>
<!ELEMENT class_elem (int_cl_name , (super | no_super) , interface_list , field_list)>
<!ELEMENT super (int_cl_name)>
<!ELEMENT no_super EMPTY>  
<!ELEMENT field_list (field ∗)>
<!ELEMENT field (arg , field_name)>
<!ELEMENT field_name (#PCDATA)>
<!ELEMENT method_section (method_elem ∗)>
<!ELEMENT method_elem (int_cl_name , method_signature , (code | no_code) , handler_list , req_meth_list)>
<!ELEMENT code (#PCDATA)>
<!ELEMENT no_code EMPTY>
<!ELEMENT handler_list (handler ∗)>
<!ELEMENT handler (int_cl_name)>
<!ELEMENT req_meth_list (req_meth ∗)>
<!ELEMENT req_meth (int_cl_name , method_signature)>
<!ELEMENT method_cfg (int_cl_name , method_signature , node_edge_list)>
<!ELEMENT node_edge_list (node_edge ∗)>
<!ELEMENT node_edge (source_node , edge , target_node)>
<!ELEMENT source_node (int_cl_name , method_signature , pc , node_label_list , (excp | no_excp) , (return | no_return))>
<!ELEMENT pc (#PCDATA)>
<!ELEMENT node_label_list (node_labels)>
<!ELEMENT node_label (#PCDATA)>
<!ELEMENT excp EMPTY>
<!ELEMENT no_excp EMPTY>
<!ELEMENT return EMPTY>
<!ELEMENT no_return EMPTY>
<!ELEMENT edge (instr , edge_label_list)>
<!ELEMENT instr (#PCDATA)>
<!ELEMENT edge_label_list (edge_label ∗)>
<!ELEMENT edge_label (#PCDATA)>
<!ELEMENT target_node (int_cl_name , method_signature , pc , node_label_list , (excp | no_excp) , (return | no_return))>
<!ELEMENT method_pp_map (called_method , method_pp_list)>
<!ELEMENT called_method (int_cl_name , method_signature)>
<!ELEMENT method_pp_list (method_pp ∗)>
<!ELEMENT method_pp (method_pp)>
<!ELEMENT calling_method (int_cl_name , method_signature)>
<!ELEMENT all_excp_prop (excp_props)>
<!ELEMENT excp_prop (int_cl_name , method_signature , excp_prop_list)>
<!ELEMENT excp_prop_list (excp_prop ∗)>
<!ELEMENT excp_prop (excp_prop)>
<!ELEMENT int_cl_name ( #PCDATA )>

Figure B.1: DTD file stored.
Bibliography


BIBLIOGRAPHY


