Provably Correct Control-Flow Graphs from Java Programs with Exceptions

Afshin Amighi[1], Pedro de Carvalho Gomes[2], and Marieke Huisman[1]

[1] University of Twente, Enschede, The Netherlands
{a.amighi,m.huisman}@utwente.nl
[2] KTH, Royal Institute of Technology, Stockholm, Sweden
pedrodcg@csc.kth.se

Abstract. We present an algorithm to extract flow graphs from Java bytecode, focusing on exceptional control flows. We prove its correctness, meaning that the behaviour of the extracted control-flow graph is an over-approximation of the behaviour of the original program. Thus any safety property that holds for the extracted control-flow graph also holds for the original program. This makes control-flow graphs suitable for performing different static analyses.
For precision and efficiency, the extraction is performed in two phases. In the first phase the program is transformed into a BIR program, where BIR is a stack-less intermediate representation of Java bytecode; in the second phase the control-flow graph is extracted from the BIR representation. To prove the correctness of the two-phase extraction, we also define a direct extraction algorithm, whose correctness can be proven immediately. Then we show that the behaviour of the control-flow graph extracted via the intermediate representation is an over-approximation of the behaviour of the directly extracted graphs, and thus of the original program.

Keywords: Software Verification, Static Analysis, Program Models

1 Introduction

Over the last decade software has become omnipresent, and in parallel, the demand for software quality and reliability has been steadily increasing. Different formal techniques are used to reach this goal, e.g., static analysis, model checking and (automated) theorem proving. A major remaining problem is that the state space of software is enormous (and often even infinite). Thus, appropriate abstractions make the formal analysis tractable. It is important for such abstractions that they are sound w.r.t. the original program: if a property holds over the abstract model, it should also be a property of the original program.

A common abstraction is to extract a program model from code, only preserving information that is relevant for the property at hand. In particular, control-flow graphs (CFGs) [4] are a widely used abstraction, where only the flow information is kept, and all program data is abstracted away. Concretely,

in a control-flow graph, the nodes represent the control points of a method, and the edges represent the instructions that make the transitions between control points. Usually, CFG is not a suitable abstraction for verifications that need data.

The literature contains several approaches to extract control-flow graphs automatically from program code. However, typically no formal argument is given why the extraction is property-preserving. This paper fills this gap: it defines a flow graph extraction algorithm for Java bytecode (JBC) *and* it proves that the extraction algorithm is sound w.r.t program behaviour. The extraction algorithm considers all the typical intricacies of Java, e.g., virtual method call resolution, the differences between dynamic and static object types, and exception handling.

The analysis of exceptional flows is a major complication to extract control-flow graphs of Java bytecode for two distinct reasons. First, the stack-based nature of the Java Virtual Machine (JVM) makes it hard to determine the type of explicitly thrown exceptions, thus making it difficult to determine to which (exceptional) control point the program will flow. Second, the JVM can raise (implicit) run-time exceptions, such as *NullPointerException* and *IndexOutOfBoundsException*; to keep track of where such exceptions can be raised requires much information. This paper covers the explicitly thrown instructions, and a significative subset of run-time exceptions.

Two different extraction algorithms are presented. The first extraction algorithm (in Section 3) creates flow graphs directly from Java bytecode. Its correctness proof is quite direct, but the resulting control-flow graph is large: in bytecode, all operands are on the stack, thus many instructions for stack manipulation are necessary, which all give rise to an *internal transfer* edge in the control-flow graph. Moreover, because the operands of a `throw` instruction are also on the stack, the exceptional control-flow is significantly over-approximated. This algorithm produces a complete map from the JBC instructions to the control-flow of the program, however, it is not so efficient for control-flow safety verifiers (e.g. to verify whether a sequence of method calls is correct).

As an alternative, we also present a two-phase extraction algorithm using the bytecode Intermediate Representation (BIR) language [5]. BIR is a stackless representation of JBC. Thus all instructions (including the explicit `throw`) are directly connected with their operands and this simplifies the analysis of explicitly thrown exceptions. Moreover the representation of a program in BIR is smaller, because operations are not stack-based, but represented as expression trees. As a result, the CFGs are efficient. BIR has been developed by Demange *et al.* as a module of SAWJA [8], a library for static analysis of Java bytecode. Demange *et al.* have proven that their translation from bytecode to BIR is semantics-preserving with respect to observable events, such as throwing exceptions and sequences of method invocations. Advantages of using the transformation into BIR are that (1) it is proven correct, and (2) it generates special assertions that indicate whether the next instruction could potentially throw a run-time exception. Our indirect extraction algorithm first generates BIR from

JBC (using the transformation of Demange *et al.*), and then our own algorithm to extract control-flow graphs from BIR.

There is no behavior definition for BIR. Therefore, to prove the correctness of the indirect extraction, we connect the BIR CFGs to the CFGs produced by the direct algorithm. We show that every BIR CFG structurally simulates the JBC CFG. This allows us to exploit an existing result that structural simulation induces behavioural simulation. Further, we prove that the CFG produced by the direct algorithm behaviourally simulates the original Java bytecode program, and from this we can conclude that the behaviours of the CFG generated by the indirect algorithm (BIR) also are a sound over-approximation of the original program behaviour. Thus, the control-flow graph extraction algorithm is sound.

Organization The remainder of this paper is organized as follows. First, Section 2 provides the necessary background definitions for the algorithm and its correctness proof. Then, Section 3 discusses the direct extraction rules for control flow graphs from Java bytecode, while Section 4 discusses the indirect extraction rules via BIR and proves its correctness. Finally Sections 5 and 6 present related work and conclude.

2 Preliminaries

2.1 Java bytecode and the Java Virtual Machine

The Java compiler translates a Java source code program into a sequence of bytecode instructions. Each instruction consists of an operation code, possibly using operands on the stack. The Java Virtual Machine (JVM) is a stack-based interpreter that executes such a Java bytecode program.

Any execution error of a Java program is reported by the JVM as an exception. Exceptions also can be thrown explicitly by instruction `athrow`. Each method can define exception handlers. If no appropriate handler can be found in the currently executing method, its execution is completed abruptly and the JVM continues looking for an appropriate handler in the caller context. This process continues until a correct handler is found or no calling context is available anymore. In the latter case, the execution terminates exceptionally.

Freund and Mitchell propose a formal framework for Java bytecode [6]. A JBC program is modeled as an environment Γ, which is a partial map from class names, interface names and method signatures to their respective definitions. Subtyping in an environment is indicated by $\Gamma \vdash \tau_1 <: \tau_2$, meaning τ_1 is a subtype of τ_2 in environment Γ. Let $\textsc{Meth}$ be a set of method signatures. A method $m \in \textsc{Meth}$ in an environment Γ is represented as $\Gamma[m] = \langle P, H \rangle$, where P denotes the body and H the exception handler table of method m. Let $\textsc{Addr}$ be the set of all valid instruction addresses in Γ. Then $Dom(P) \subset \textsc{Addr}$ is the set of valid program addresses for method m and $P[k]$ denotes the instruction at position $k \in Dom(P)$ in the method's body. For convenience, $m[k] = i$ denotes instruction $i \in Dom(P)$ at location k of method m.

In this formal model, a JVM execution state is a configuration $C = A; h$, where A denotes the sequence of activation records and h is the heap. Each activation record is created by a method invocation. Formally the sequence is defined as follows:

$$A \ ::= \ A' \mid \langle x \rangle_{exc}.A' \quad ; \quad A' \ ::= \ \langle m, pc, f, s, z \rangle.A' \mid \epsilon$$

Here, m is the method signature of the active method, pc is the program counter, f is a map from local variables to values, s is the operand stack, and z is initialization information for the object being initialized in a constructor. Finally, $\langle x \rangle_{exc}$ is an exception handling record, where $x \in \textsc{Excp}$ denotes the exception: in case of an exception, the JVM pushes such a record on the stack.

To handle exceptions, the JVM searches the exception table declared in the current method to find a corresponding set of instructions. The method's exception table H is a partial map which has the form $\langle b, e, t, \varrho \rangle$, where $b, e, t \in \textsc{Addr}$ and $\varrho \in \textsc{Excp}$. If an exception of subtype ϱ in environment Γ is thrown by an instruction with index $i \in [b, e)$ then $m[t]$ will be the first instruction of the corresponding handler. Thus, the instructions between b and e model the `try` block, while the instructions starting at t model the `catch` block that handles the exception. In order to manage `finally` blocks, a special type of exception called *Any* is defined. The instructions in a finally block always have to be executed by the JVM, therefore all exceptions are defined as a subtype of *Any*.

2.2 Program Model

Control-flow graphs present an abstract model of a program. To define the structure and behavior of a control-flow graph we follow Gurov et al. and use the general notion of *model* [7, 9].

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

Method specifications are the basic building blocks of flow graphs. To model sequential programs with procedures and exceptions, method specifications are defined as an instantiation of initialized models as follows.

Definition 2 (Method Specification). *A* flow graph with exceptions *for* $m \in \textsc{Meth}$ *over sets* $M \subseteq \textsc{Meth}$ *and* $E \subseteq \textsc{Excp}$ *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 *the set of the labels which can be any instantiation to indicate the 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.* $\mathbb{E}_m \subseteq V_M$ *is a non-empty set of entry control point(s) of* m*.*

A node $v \in V_m$ is marked with atomic proposition r to indicate that it is a return node of the method. The labeling set L_m is not specified intentionally to accept any instantiation. For example, figure 1 shows a sample program with corresponding CFG in which on the contrary to internal transitions, method calls are important. So L_m is instantiated as $L_m = M \cup \{\varepsilon\}$.

```
public class Number {
   public static boolean even(int n){
      if(n==0)
         return true;
      else
         return odd(n-1);
   }
   public static boolean odd(int n){
      if(n==0)
         return false;
      else
         return even(n-1);
   }
}
```

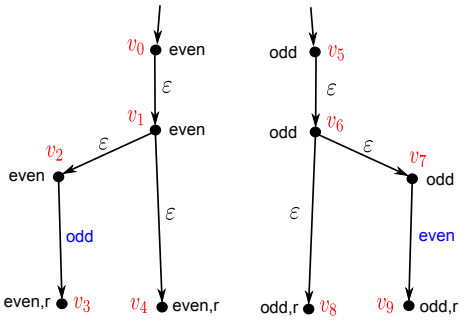

Fig. 1. Method specifications of methods `even` and `odd`

Every flow graph comes with an interface that defines which methods are provided to and required from the environment.

Definition 3 (Flow Graph Interface). *A* flow graph interface *is a triple* $I = (I^+, I^-, E, M_e)$*, where* $I^+, I^- \subseteq$ Meth *are finite sets of provided and required method signatures, and* $E \subseteq$ Excp *is a finite set of exceptions and* $M_e \subseteq$ Meth *is the set of entry methods (starting points of the program), respectively. If* $I^- \subseteq I^+$ *then* I *is* closed.

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

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

- $(\mathcal{M}_m, \mathbb{E}_m) : (\{m\}, \mathcal{M}, E)$ *if* $(\mathcal{M}_m, \mathbb{E}_m)$ *is a method specification for* m *over* M *and* E*,*
- $\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ *if* $\mathcal{G}_1 : I_1$ *and* $\mathcal{G}_2 : I_2$*.*

Definition 5 (Weak Simulation over Models). *Let* $a \Longrightarrow b$ *be a sequence of silent transitions (possible zero) from* a *to* b*, and* $a \overset{\beta}{\Longrightarrow} b$ *a sequence containing a single visible transition* β*, and zero to many silent transitions. A weak simulation over a model* $(S, L, \rightarrow, A, \lambda)$ *is a binary relation* R_w *over* S $(R_w \subseteq S \times S)$ *such that* $\forall p, q \in S$*, if* $(p, q) \in R_w$ *then* $\lambda(p) = \lambda(q)$ *and* $\forall \beta \in L, \forall p' \in S$*, if* $p \overset{\beta}{\Longrightarrow} p'$ *implies that there is a* $q' \in S$ *such that* $q \overset{\beta}{\Longrightarrow} q'$ *and* $(p', q') \in R_w$*.*

We use the following proposition to prove the weak simulation relation. The proof is trivial and we omit it.

Proposition 1. *R_w is a weak simulation if and only if $\forall (p,q) \in R_w$*
if $p \rightarrow p'$ then exists $q' \in S$ such that $q \implies q' \wedge (p',q') \in R_w$.
if $p \xrightarrow{\beta} p'$ then exists $q' \in S$ such that $q \overset{\beta}{\implies} q' \wedge (p',q') \in R_w$.

3 Extracting Control-Flow graphs from bytecode

CFG construction rules use bytecode instructions to build the graph. Depending on the instruction at a given address, edges between the current control node and the possible next control nodes are constructed.

For convenience, we group the different JBC instructions into disjoint sets: RetInst is the set of normal return instructions (e.g. `return`); CmpInst is the set of simple computational instructions (e.g. `push v, pop`); CndInst is the set of conditional instructions (e.g. `ifeq q`); JmpInst is the set of jump instructions (e.g. `goto q`); XmpInst is the set of instructions that potentially can raise an exception (e.g. `div, getfield f`); InvInst is the set of method invocation instructions (e.g. `invokevirtual (o,m)`); and ThrInst = {`throw` X}, where instruction `throw X` is the result of a stack analysis of the JBC. In JBC, `athrow` does not accept any argument and the type of the exception is determined at run-time (as the top of the stack). Stack analysis of the JBC can generate an exception variable to be thrown at run-time. We over-approximate the type of the exception using a set X that contains the static type of the variable, which is the result of the stack analysis and all its subtypes.

We define a JBC method body as a sequence of address and instruction pairs:

$$S ::= \quad \ell : inst \ ; \ S \mid \epsilon \quad \ell \in \textsc{Addr},\ inst \in \textsc{Inst}$$

The nodes in a method CFG, define a map of the method's execution state, covering all possible JVM configurations. All nodes are tagged with pairs of an address and a method signature. The set of the addresses is extended by adding symbol $\natural$ to denote the *abort* state[3] of a program. Based on Definition 2, to construct the nodes we have to specify V_m, A_m and λ_m. For a node $v \in V_m$ indicating control point $\ell \in \textsc{Addr}_\natural$ of method m, we define $v = (m, \ell)$. The labeling function λ_m specifies A_m for a given $v \in V_m$. If $m[\ell] \in$ RetInst then the node is tagged with r. If the node is an exceptional node (an exception is raised) then it is marked with the exception type $x \in E$. The corresponding method signature is the default tag for all the method's control nodes. If $\ell = 0$ then the node will be a member of $\mathbb{E}_m$.

Two nodes are equal if they specify the same control address of the same method with equal atomic proposition sets. We use the following notation: $v \vDash x$

[3] The JVM's attempt to find a proper handler for an exception is unsuccessful and the program terminates abnormally.

$$
\begin{array}{l}
m\mathcal{G}(S_1; S_2, H) = m\mathcal{G}(S_1, H) \ \cup \ m\mathcal{G}(S_2, H) \\
m\mathcal{G}((p,i),H) = \{(\circ^p_m, i_g, \circ^{succ\ p}_m)\} \quad \texttt{if}\ i \in \textsc{CmpInst} \\
m\mathcal{G}((p,i),H) = \{(\circ^p_m, i_g, \circ^{q}_m)\} \quad \texttt{if}\ i \in \textsc{JmpInst} \\
m\mathcal{G}((p,i),H) = \{(\circ^p_m, i_g, \circ^{succ\ p}_m), (\circ^p_m, i_g, \circ^{q}_m)\} \quad \texttt{if}\ i \in \textsc{CndInst} \\
m\mathcal{G}((p,i),H) = \left\{\{(\circ^p_m, i_g, \bullet^{p,x}_m)\} \ \cup \ \mathcal{H}^x_p \mid x \in X\right\} \quad \texttt{if}\ i = \texttt{throw}\ X \\
m\mathcal{G}((p,i),H) = \{(\circ^p_m, i_g, \circ^{succ\ p}_m)\} \ \cup \ \mathcal{E}^i_p \quad \texttt{if}\ i \in \textsc{XmpInst} \\
\mathcal{E}^i_p = \left\{\{(\circ^p_m, i_g, \bullet^{p,x}_m)\} \ \cup \ \mathcal{H}^x_p \mid x \in \mathcal{X}(i)\right\} \\
m\mathcal{G}((p,i),H) = \{(\circ^p_m, i_g, \bullet^{p,\varrho_N}_m)\} \ \cup \ \mathcal{R}^i_p \cup \ \mathcal{H}^{\varrho_N}_p \cup \ \mathcal{N}^x_p \quad \texttt{if}\ i \in \textsc{InvInst} \\
\mathcal{R}^i_p = \{(\circ^p_m, call\ (\tau, n), \circ^{succ\ p}_m) \mid \ \tau \in Rec_\Gamma(i)\} \\
\mathcal{N}^x_p = \left\{\{(\circ^p_m, handle_n, \bullet^{p,x}_m)\} \ \cup \ \mathcal{H}^x_p \mid \ \bullet^{q,x,r}_n \in \ m\mathcal{G}(n),\ n \in \ res^\alpha_\Gamma(o,n)\ \right\} \\
(\ \Gamma \vdash x <: y\ ) \implies \mathcal{H}^x_p = \begin{cases} \{(\bullet^{p,x}_m, handle, \circ^t_m)\} & \hbar_{\Gamma[m]}(p,y) = t \neq 0 \\ \{(\bullet^{p,x}_m, handle, \bullet^{p,x,r}_m)\} & \hbar_{\Gamma[m]}(p,y) = 0 \ \wedge \ m \notin M_e \\ \{(\bullet^{p,x}_m, handle, \bullet^{b,x,r}_m)\} & \hbar_{\Gamma[m]}(p,y) = 0 \ \wedge \ m \in M_e \end{cases}
\end{array}
$$

Fig. 2. CFG Construction Rules

means that node v is tagged with exception x; $\bullet^{\ell,x}_m$ indicates an exceptional control node and $\circ^\ell_m$ denotes a normal control node.

The CFG extraction rules for method m in environment Γ use the implementation of the method, $\Gamma[m] = \langle P, H\rangle$. For each instruction in $\Gamma[m]$, the rules build a set of labeled edges connecting control nodes.

Definition 6 (Method Control-Flow Graph Extraction). *Let V be the set of nodes and $I_g = \textsc{Inst}_g \cup \{handle\}$, where $\textsc{Inst}_g$ is any mapping from* Inst *to the corresponding instruction. Let Π be a set of environments. Then the control-flow graph extraction of method m is $m\mathcal{G} : \Pi \times \textsc{Meth} \to \mathcal{P}(V \times I_g \times V)$, defined in Figure 2 (where succ denotes the next instruction address function).*

The construction rules are defined purely syntactically, based on the method's instructions. However, intuitively they justify the instruction's operational semantics. The first rule decomposes a sequence of instructions into individual instructions. For each individual instruction, a set of edges is computed.

For simple computational instructions, a direct edge to the next control address is produced. For jump instructions, an edge to the jump address (q, specified in the instruction) is generated. For conditional instructions two edges are generated: to the next control address and to the address specified for the jump (q). For instructions in XmpInst edges for all possible flows are added: successful execution, and exceptional execution, with edges for successful and failed exception handling, as defined by function $\mathcal{H}^x_p$. This function constructs the outgoing edges of the exceptional nodes by searching the exception table for a suitable handler of exception type x at position p. If there is such a handler, it returns a transition edge from an exceptional node to a normal node. Otherwise it produces a transition to an exceptional return node. Function $\hbar$ seeks the proper handler in the exception handling table; it returns 0 if there is no entry for the exception at the specified control point. The function $\mathcal{X} : \textsc{XmpInst} \to \mathcal{P}(\textsc{Excp})$

is to determine possible exceptions of a given instruction. The `throw` instruction is handled similarly, where X is the set of possible exceptions, identified by the transformation algorithm.

To extract edges for method invocations, function $Rec_\Gamma(i)$ determines the set of possible receivers of a method call in environment Γ. For `invokevirtual`, the receiver is determined by late binding, and the virtual method call (VMC) resolution function res^α_Γ will be used, where α is a standard static analysis technique to resolve VMC.

$$Rec_\Gamma(i) = \begin{cases} \{staticT(o)\} \text{ if } i \in \{\texttt{invokespecial (o,n)}, \texttt{invokestatic (o,n)}\} \\ res^\alpha_\Gamma(o,n) \quad \text{if } i = \texttt{invokevirtual (o,n)} \end{cases}$$

Suppose that VMC resolution uses the *RTA* algorithm, i.e., $\alpha = RTA$, then the result of the resolution for object o and method n in environment Γ will be:

$$res^\alpha_\Gamma(o,n) = \{\tau \mid \tau \in IC_\Gamma \ \wedge \ \Gamma \vdash \tau <: staticT(o) \ \wedge \ n = lookup(n,\tau)\}$$

where IC_Γ is the set of instantiated classes in environment Γ, $staticT(o)$ gives the static type of object o and $lookup(n,\tau)$ corresponds to the signature of n, i.e., τ is a subtype of o's static type and method n is defined in class τ.

Given the set of possible receivers, calls are generated for each possible receiver. For each call, if the method's execution terminates normally, control will be given back to the next instruction of the caller. If the method terminates with an uncaught exception, the caller has to handle this propagated exception. If the current method is an entry method, m_e, then the program will terminate abnormally. The CFG extraction rules for method invocations produce edges for both ϱ_N=*NullPointerException* and for all propagated exceptions.

$\mathcal{R}^i_p$ is the set of the edges for normal terminating calls, $\mathcal{H}^{\varrho_N}_p$ is the set of edges to handle ϱ_N, and $\mathcal{N}^x_p$ defines the set of edges to handle all uncaught exceptions from all possible callees. We put the callees signature as an index of the `handle` label to differentiate between propagated exceptions from method calls and exceptions raised in the current method. Similar to generating outgoing edges for exceptional control points, $\mathcal{H}^x_p$ generates edges for successful/failed handlers for all exceptional nodes in CFG_n which is the CFG of method $n \in res^\alpha_\Gamma(o,n)$.

The CFG of a Java class C, denoted $c\mathcal{G}(C) : \textsc{Class} \to \mathcal{P}(V \times \textsc{Inst}_g \times V)$, is defined as the disjoint union of the CFGs of the methods in C. The CFG of a program P, denoted $\mathcal{G}(P) : \Pi \to \mathcal{P}(V \times \textsc{Inst}_g \times V)$, is the disjoint union of all CFGs of the classes in P.

3.1 CFG Correctness

In order to prove the soundness of the extracted flow graph we need to define the behavior of the flow graph. The following extends the behavior definition of flow graphs from [9], based on our extraction rules.

Definition 7 (CFG Behavior). *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed flow graph with exceptions such that $\mathcal{M} = (V, L, \to, A, \lambda)$. The* behavior *of $\mathcal{G}$ is described by the specification $b(\mathcal{G})$, where $\mathcal{M}_g = (S_g, L_g, \to_g, A_g, \lambda_g)$ 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* $L_g^C = \{m_1\ l\ m_2 \mid l \in \{call, ret, xret\}, m_1, m_2 \in I^+\}$ *(the set of call and return labels) and* $L_g^X = \{l\ x \mid l \in \{throw,\ catch\}, x \in \textsc{Excp}\}$ *(the set of exceptional transition labels).*
- $A_g = A$ *and* $\lambda_g((v, \sigma)) = \lambda(v)$
- $\rightarrow_g \subset S_g \times S_g$ *is the set of transitions in* CFG_m *with the following rules:*

[call] $(v_1, \sigma) \xrightarrow{m_1\ call\ m_2}_g (v_2, v_1.\sigma)$ *if* $m_1, m_2 \in I^+, v_1 \xrightarrow{call\ m_2}_{m_1} v_1',\ v_1' \in next(v_1),\ v_1 \not\models \textsc{Excp},\ v_2 \models m_2,\ v_2 \in \mathbb{E},\ v_1 \models \neg r$

[return] $(v_2, v_1.\sigma) \xrightarrow{m_2\ ret\ m_1}_g (v_1', \sigma)$ *if* $m_1, m_2 \in I^+, v_2 \models m_2 \wedge r,\ v_1 \models m_1,\ v_1' \not\models \textsc{Excp},\ v_1' \in next(v_1)$

[xreturn] $(v_2, v_1.\sigma) \xrightarrow{m_2\ xret\ m_1}_g (v_1', \sigma)$ *if* $m_1, m_2 \in I^+,\ v_2 \models m_2,\ v_1 \models m_1,\ v_2 \xrightarrow{handle}_{m_2} v_2',\ v_1 \xrightarrow{handle}_{m_1} v_1',\ v_2 \models x,\ v_2' \models x \wedge r,\ v_1 \not\models x,\ v_1' \models x, x \in \textsc{Excp}$

[transfer] $(v, \sigma) \xrightarrow{\tau}_g (v', \sigma)$ *if* $m \in I^+,\ v \xrightarrow{ig}_m v', v \models \neg r,\ v \not\models \textsc{Excp},\ v' \not\models \textsc{Excp}$

[throw] $(v, \sigma) \xrightarrow{throw\ x}_g (v', \sigma)$ *if* $m \in I^+, v \xrightarrow{ig}_m v', v \models \neg r, v' \models \textsc{Excp}$

[catch] $(v, \sigma) \xrightarrow{catch\ x}_g (v', \sigma)$ *if* $m \in I^+, v \xrightarrow{handle}_m v', v \models \neg r \wedge \textsc{Excp}, v' \not\models r, v' \not\models \textsc{Excp}$

Consider again the flow graph in Figure 1. One example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{even\ call\ odd} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\tau} (v_8, v_3) \xrightarrow{odd\ ret\ even} (v_3, \varepsilon)$$

To show the correctness of the extraction algorithm, we show that the extracted CFG of method m can match all possible moves during execution of m. In order to do this, we first define a mapping θ that abstracts JVM configurations to CFG behavioural configurations. Using θ, we can then prove that the behaviour of a CFG simulates the behaviour of the corresponding method in JBC.

Definition 8 (Abstraction Function for VM States). *Let* $\textsc{Vmc}$ *be the set of JVM execution configurations and* S_g *the set of states in* $m\mathcal{G}$*. Then* $\theta : \textsc{Vmc} \rightarrow S_g$ *is defined inductively as follows:*

$$\theta(\langle m, p, f, s, z\rangle.A; h) = \langle \circ_m^p, \theta(A; h)\rangle \qquad \theta(\langle x\rangle_{exc}.\epsilon; h) = \langle \bullet_m^{\flat, x, r}, \epsilon\rangle$$
$$\theta(\langle m, p, f, s, z\rangle.\epsilon; h) = \langle \circ_m^p, \epsilon\rangle \qquad \theta(\langle x\rangle_{exc}.\langle m, p, f, s, z\rangle.A; h) = \langle \bullet_m^{p, x}, \theta(A; h)\rangle$$

Now we can prove correctness of the CFG construction. Function θ specifies the corresponding JVM state in the extracted CFG. In order to match relating transitions we use simulation modulo relabeling: we map JVM transition labels $\textsc{Inst} \cup \{\epsilon\}$ to the CFG transition labels in CFG $\textsc{Inst}_g \cup \{handle\}$. Transition ϵ in the JVM labeling set denotes silent transitions: transitions of the JVM to handle raised exceptions.

$$
\begin{array}{rll}
\mathit{expr} ::= & c \mid \texttt{null} & \text{(constants)} \\
& \mid \mathit{expr} \oplus \mathit{expr} & \text{(arithmetic)} \\
& \mid \mathit{tvar} \mid \mathit{lvar} & \text{(variables)} \\
& \mid \mathit{expr}.\mathtt{f} & \text{(field access)} \\
\mathit{lvar} ::= & \mathtt{l} \mid \mathtt{l}_1 \mid \mathtt{l}_2 \mid \ldots \mid \texttt{this} & \text{(local var.)} \\
\mathit{tvar} ::= & \mathtt{t} \mid \mathtt{t}_1 \mid \mathtt{t}_2 \mid \ldots & \text{(temp. var.)} \\
\mathit{target} ::= & \mathit{lvar} & \\
& \mid \mathit{tvar} & \\
& \mid \mathit{expr}.\mathtt{f} &
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Assignment} ::= & \mathit{lvar}\ \texttt{:=}\ \mathit{expr} \mid \mathit{expr}.\mathtt{f}\ \texttt{:=}\ \mathit{expr} \\
\mathit{TempAssign} ::= & \mathit{tvar}\ \texttt{:=}\ \mathit{expr} \\
\mathit{Return} ::= & \texttt{vreturn}\ \mathit{expr} \mid \texttt{return} \\
\mathit{MethodCall} ::= & \mathit{expr}.\mathtt{m}(\mathit{expr}, \ldots, \mathit{expr}) \\
& \mid \mathit{target} := \mathit{expr}.\mathtt{m}(\mathit{expr}, \ldots, \mathit{expr}) \\
\mathit{NewObject} ::= & \mathit{target} := \texttt{new C}(\mathit{expr}, \ldots, \mathit{expr}) \\
\mathit{Assertion} ::= & \texttt{notnull}\ \mathit{expr} \mid \texttt{notzero}\ \mathit{expr} \\
\\
\mathit{instr} ::= & \texttt{nop} \mid \texttt{if}\ \mathit{expr}\ \texttt{pc} \mid \texttt{goto pc} \\
& \mid \texttt{throw}\ \mathit{expr} \mid \texttt{mayinit C} \\
& \mid \mathit{Assignment} \mid \mathit{TempAssign} \\
& \mid \mathit{Return} \mid \mathit{MethodCall} \\
& \mid \mathit{NewObject} \mid \mathit{Assertion}
\end{array}
$$

Fig. 3. Expressions and Instructions of BIR

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

Proof. For every possible JVM configuration c and instruction i, we establish the possible transitions to a set of configurations C based on the operational semantics. We apply θ to all elements in C, denoted $\Theta(C)$, to determine the abstract CFG configurations. Then we use the CFG construction algorithm to determine which edges are established for instruction i. These edges determine the possible transitions paths from $\theta(c)$ to the next CFG states S, and we show that the set S corresponds to the configurations $\Theta(C)$. To show that this indeed holds, we use a case analysis on VMC. For more details we refer to Amighi's Master thesis [2]. □

4 Extracting Control-Flow Graphs from BIR

This section presents the two-phase transformation from Java bytecode into control-flow graphs using BIR as intermediate representation. First we briefly present BIR and the transformation from JBC into BIR. Then, we present the transformation from BIR into control-flow graphs and prove its correctness.

4.1 The BIR language

The BIR language is an intermediate representation of Java bytecode. The main difference with standard JBC is that BIR instructions are stack-less, i.e., they have explicit operators and do not operate over values stored in the operand stack. This subsection gives a brief overview of BIR, for a full account we refer to [5].

Syntax and Expression trees Figure 3 summarizes the BIR syntax. Its instructions operate over expression trees, i.e., arithmetic expressions composed of constants, operations, variables, and fields of other expressions (*expr*.f). BIR does

not have operations over strings and booleans,these are transformed into methods calls by the `BC2BIR` transformation. The transformation algorithm discussed below reconstructs expression trees, i.e., it collapses one-to-many stack-based operations into a single expression. As a result, a program represented in BIR typically has fewer instructions than the original JBC program.

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

Many of the BIR instructions have an equivalent JBC counterpart, e.g., `nop`, `goto` and `if`. A `vreturn` *expr* ends the execution of a method with return value *expr*, while `return` ends a *void* method. The `throw` instruction explicitly transfers control flow to the exception handling mechanism. Method call instructions are represented by the method signature. For non-*void* methods, the instruction assigns the result value to a variable.

In contrast to JBC, object allocation and initialization happen in a single step, during the execution of the `new` instruction. However, Java also has class initialization, i.e., the one-time initialization of a class's static fields. To preserve this class initialization order, BIR contains a special `mayinit` instruction. This behaves exactly as a `nop`, but indicates that at that point a class may be initialized for the first time.

Assertions The support for run-time exceptions in BIR is implemented in the form of special instructions called assertions. These instructions are inserted during the transformation of bytecode instructions that can potentially raise exceptions, as defined in the Java Virtual Machine specification.

We define $\mathcal{RE}$ as the set of supported run-time exceptions in BIR (following [3]). Figure 4 shows this set, and the function $\bar{\chi} : \mathit{Assertion} \rightarrow \mathcal{RE}$ that maps the assertion to the run-time exception it guards. Along the text we exemplify the use of assertions using `[notnull]` and `[notzero]` only, and its corresponding exceptions.

Assertion	$\mathcal{RE}$
`[notnull]`	*NullPointerException*
`[checkbound]`	*IndexOutOfBoundsException*
`[notneg]`	*NegativeArraySizeException*

Assertion	$\mathcal{RE}$
`[notzero]`	*ArithmeticException*
`[checkcast]`	*ClassCastException*
`[checkstore]`	*ArrayStoreException*

Fig. 4. $\bar{\chi}$: Mapping of Assertions and Runtime Exceptions

The `[notzero]` *expr* assertion is placed before all instructions containing an expression with division operation. It checks whether the divisor *expr* evaluates to zero, thus potentially raising an *ArithmeticException*. The `[notnull]` *expr* assertion is placed before any access to a reference and checks whether `expr` evaluates to a dereferenced object, thus raising a *NullPointerException*. In cases the assertion is successful, it behaves as a `[nop]`, and control-flow passes to

the next instruction. In case of a failure, control is transferred to the exception handling mechanism, just like for a `[throw]` instruction. If a suitable exception handler is found, control is moved to the first instruction of this handler.

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

4.2 Transformation from Java bytecode into BIR

Next we give a short overview of the `BC2BIR` transformation. In some points, the algorithm is quite complex, because it has to maintain consistency between object references and BIR variables. However, since the flow-graph extraction abstracts away from all data, these complex points are not relevant and we do not discuss them here. Instead we focus on the transformation of instructions, i.e., the $\texttt{BC2BIR}_{instr}$ function. For the complete algorithm, we refer to [5].

The transformation `BC2BIR` transforms a complete JBC program into BIR by symbolically executing the bytecode using an abstract stack. This stack is used to reconstruct expression trees. Moreover, it also stores references to uninitialized objects, used to correctly match them with the corresponding initialization instruction, and differentiate the to constructor of the super class.

Definition 9 (Abstract Stack). *Let* $\mathtt{UR} = \{\mathtt{UR}^{\mathtt{C}}_{\mathtt{pc}} | \mathtt{C} \in \mathbb{C}, \mathtt{pc} \in \mathbb{N}\}$ *be the set of references to uninitialized objects with static type* `C`*, allocated at program counter* `pc`*. Let Expr be the set of expression trees in the BIR language. Then the* abstract stack *is defined as*

$$AbsStack = (Expr \cup \ \mathtt{UR})*$$

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

Definition 10 (BIR Transformation Function). *The rules defining the* instruction-wise transformation $\texttt{BC2BIR}_{instr} : \mathbb{N} \times instr \times AbsStack \rightarrow (instr_{BIR} * \times AbsStack) \cup Fail$ *from Java bytecode into BIR are given in Figure 5.*

As a remark, JBC instructions with similar semantics, but working on different types of operands (e.g., `adiv` and `fdiv`) are grouped as single instructions (e.g., `div`). As a convention, we use brackets to distinguish BIR instructions from their JBC counterpart. At several places, the transformation function introduces new variables $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ that maintain consistency between values on the stack and the value that it represents.

Input		Output	
Instr	Stack	Instrs	Stack
`nop`	as	$\emptyset$	as
`pop`	`e`:as	$\emptyset$	as
`push c`	as	$\emptyset$	`c`:as
`dup`	`e`:as	$\emptyset$	`e`:`e`:as
`load x`	as	$\emptyset$	`x`:as
`add`	$\mathtt{e}_1$:$\mathtt{e}_2$:as	$\emptyset$	$\mathtt{e}_1$+$\mathtt{e}_2$:as
`div`	$\mathtt{e}_1$:$\mathtt{e}_2$:as	`[notzero` $\mathtt{e}_2$`]`	$\mathtt{e}_1$/$\mathtt{e}_2$::as

Input		Output	
Instr	Stack	Instrs	Stack
`if pc'`	`e`:as	`[if e pc']`	as
`goto pc'`	as	`[goto pc']`	as
`return`	as	`[return]`	as
`vreturn`	`e`:as	`[return e]`	as
`athrow`	`e`:as	`[throw e]`	as
`new C`	as	`[mayinit C]`	$UR^{\mathtt{C}}_{\mathtt{pc}}$:$as$
`getfield f`	`e`:as	`[notnull e]`	`e:f`:as

Input		Output		Condition
Instr	Stack	Instrs	Stack	
`store x`	`e`:as	`[x:=e]`	as	`x` $\notin as$
		$[\mathtt{t}^0_{\mathtt{pc}}$`:=x;x:=e]`	$as[\mathtt{t}^0_{\mathtt{pc}}/\mathtt{x}]$	`x` $\in as$
`putfield f`	$\mathtt{e}'$:`e`:as	`[notnull e;`$FSave$`(pc,f,`as`);e.f:=`$\mathtt{e}'$`]`	$as[\mathtt{t}^i_{\mathtt{pc}}/\mathtt{e}_i]$	
`invokevirtual m` $\mathtt{e}'_1 ... \mathtt{e}'_n$`:e:`$as$		`[notnull e;`$Hsave$`(pc,`as`)]`		
		$[\mathtt{e.m}(\mathtt{e}'_1 ... \mathtt{e}'_n)]$	$as[\mathtt{t}^j_{\mathtt{pc}}/\mathtt{e}_j]$	`m` is *void*
		$[\mathtt{t}^0_{\mathtt{pc}}:=\mathtt{e.m}(\mathtt{e}'_1 ... \mathtt{e}'_n)]$	$\mathtt{t}^0_{\mathtt{pc}}$:$as[\mathtt{t}^j_{\mathtt{pc}}/\mathtt{e}_j]$	`m` not *void*
`invokespecial m` $\mathtt{e}'_1 ... \mathtt{e}'_n$`:e:`$as$		$[Hsave(\mathtt{pc},as);\mathtt{t}^0_{\mathtt{pc}}:=\mathtt{new\ C}(\mathtt{e}'_1 ... \mathtt{e}'_n)]$	$as[\mathtt{t}^j_{\mathtt{pc}}/\mathtt{e}_j]$	`e` $= UR^{\mathtt{C}}_{\mathtt{pc}}$
		$[\mathtt{notnull\ e};Hsave(\mathtt{pc},as);\mathtt{e.m}(\mathtt{e}'_1 ... \mathtt{e}'_n)]$	$as[\mathtt{t}^j_{\mathtt{pc}}/\mathtt{e}_j]$	otherwise

Fig. 5. Rules for $\mathtt{BC2BIR}_{instr}$

JBC instructions `if`, `goto`, `return` and `vreturn` are transformed into corresponding BIR instructions (using the top of the stack as condition argument for the `if` instruction). The `new` instruction adds an unallocated object on the stack, and produces a `mayinit` instruction. The `getfield f` instruction reads a field from the object reference at the top of the stack. This might produce a *NullPointerException*, thus the transformation produces a `notnull` instruction.

For the `store x` instruction there are two cases. If the variable `x` is not yet on the stack, the assignment of the expression on the top of the stack to `x` is returned. Otherwise, first the current value of `x` is assigned to a newly created variable $\mathtt{t}^0_{\mathtt{pc}}$, and all occurrences of `x` on the stack are replaced by this new variable (denoted $as[\mathtt{t}^0_{\mathtt{pc}}/\mathtt{x}]$).

The `putfield f` outputs a set of BIR instructions: first, a `notnull` assertion, to check if the accessed reference is made to a valid object. Then the auxiliary function $FSave$ introduces a set of Assignment instructions to temporary variables, for all occurrences of `f` on the stack; finally it creates the assignment instruction to the field (`e.f`).

The rule for virtual method calls (`invokevirtual`) generates a sequence of instructions. First there is a `[notnull]` assertion. Then any reference to objects on the stack that access the heap must be stored into newly introduced variables to remember its value, because objects on the heap can be altered during the method invocation. This is defined as function *Hsave*. Finally, there is the call instruction itself. If the method returns a value, a new variable is introduced to store the return value, and this is added to the abstract stack.

The transformation of `invokespecial` searches for an uninitialized reference on the stack after the method arguments to check if such call targets an object

constructor. If such reference is not found , the transformation acts similarly to the case of virtual method calls. However if an uninitialized reference is found, it replaces the $UR^{\mathtt{C}}_{\mathtt{pc}}$ reference to the uninitialized object – added by the transformation of the `new` instruction – with a new variable $\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}$.

Figure 6 shows the JBC and BIR representations for the method `even`, presented in Figure 1. The example contains both a local variable (`$bcvar1`) and a new variable introduced by the transformation (`$irvar1`). We can observe reconstructed expression trees as the argument to the method invocation, and as the operand to the `[if]` instruction. The `[notnull this]` instruction is trivial, since it checks if the reference to the current object is valid, but it illustrates how assertions are placed before instructions that can raise exceptions.

Java bytecode	BIR
public boolean even(int);	public bool even(int)
0: iload_1	0: `if ($bcvar1 != 0) goto 2`
1: ifne 6	1: `vreturn 1`
4: iconst_1	2: `notnull this`
5: ireturn	3: `$irvar1 := this.odd($bcvar1-1)`
6: aload_0	4: `vreturn $irvar1`
7: iload_1	
8: iconst_1	
9: isub	
10: invokevirtual boolean odd(int)	
13: ireturn	

Fig. 6. Comparison of method in JBC and BIR

4.3 Transformation from BIR into Control-Flow Graphs

The setup of the extraction algorithm is similar to that of `BC2BIR`. It iterates over the instructions of a method, using the transformation function $\mathtt{b}\mathcal{G}$. Each iteration outputs a set of triples of the form $V \times Instr \times V$. The extraction algorithm $\mathtt{b}\mathcal{G}$ takes as input a program counter and an instruction array for a BIR method. It outputs a set of edges. The set of edges can then be directly transformed in a control-flow graph as defined in Definition 6.

To define $\mathtt{b}\mathcal{G}$, we introduce auxiliary functions and definitions similar to the ones introduced in the direct extraction (in Section 3). $\bar{H}$ is the exceptions table from a given method. It contains the same entries as the JBC table, but has its control points translated to the BIR. The function $\hbar_{\bar{H}}(pc, x)$ searches for the first handler for the exception x (or a subtype) at position pc. The function $res^{\alpha}_{b}(o, n)$ returns all possible receivers for a method call, given the object reference and the method signature. The function $\bar{\mathcal{H}}^{pc}_{x}$ returns an edge after querying $\hbar$ for exception handlers. Also, $\bar{\mathcal{N}}^{pc}_{n}$ returns edges to exceptional flows for the method invocations that can terminate due to an uncaught exception, and consequently propagate it.

$$\bar{\mathcal{H}}_x^{pc} = \begin{cases} (\bullet_m^{\mathtt{pc},x}, handle, \circ_m^{t}) & \hbar_{\bar{H}} = t \neq 0 \\ (\bullet_m^{\mathtt{pc},x}, handle, \bullet_m^{\mathtt{pc},x,r}) & \hbar_{\bar{H}} = 0 \end{cases}$$

$$\bar{\mathcal{N}}_n^{pc} = \{(\circ_m^{\mathtt{pc}}, handle_n, \bullet_m^{\mathtt{pc},x}), \bar{\mathcal{H}}_x^{pc} \mid \bullet_n^{pc,x,r} \in \mathtt{b}\mathcal{G}(n), n \in res_b^{\alpha}(o,n)\}$$

$$\begin{array}{lll}
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \emptyset & & \text{if } i \in \mathit{TempAssign} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}})\} & & \text{if } i \in \{\texttt{[nop]}, \texttt{[mayinit]}\} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}})\} & & \text{if } i \in \mathit{Assignment} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}}), \circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc'}})\} & & \text{if } i = \texttt{[if } \mathit{expr}\, \mathtt{pc'}\texttt{]} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc'}})\} & & \text{if } i = \texttt{[goto pc']} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc},r})\} & & \text{if } i \in \mathit{Return} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, call(\tau,n), \circ_m^{\mathtt{pc+1}}), (\circ_m^{\mathtt{pc}}, i_b, \bullet_m^{\mathtt{pc},\varrho N})\} \cup & & \text{if } i \in \mathit{NewObject} \\
\quad \{\bar{\mathcal{H}}_{\varrho N}^{pc}\} \cup \bar{\mathcal{N}}_{pc}^{n} & & \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, call(\tau,n), \circ_m^{\mathtt{pc+1}}) \mid \forall \tau \in res_b^{\alpha}\} \cup \bar{\mathcal{N}}_{pc}^{n} & & \text{if } i \in \mathit{MethodCall} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \bullet_m^{\mathtt{pc},x}), \bar{\mathcal{H}}_x^{pc} \mid x \in X\} & & \text{if } i = \texttt{[throw } X\texttt{]} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}}), (\circ_m^{\mathtt{pc}}, i_b, \bullet_m^{\mathtt{pc},\bar{\chi}(i)}), \bar{\mathcal{H}}_{\bar{\chi}(i)}^{\mathtt{pc}}\} & & \text{if } i \in \mathit{Assertion}
\end{array}$$

Fig. 7. Extraction rules for Control-flow graphs from BIR

Definition 11 (Control Flow Graph Extraction). *The* control-flow graph extraction function $\mathtt{b}\mathcal{G} : (\mathbb{N} \times \mathit{Instr}) \times \bar{H} \to \mathcal{P}((V, I_b, V))$ *is defined by the rules in Figure 7, where* $I_b = \mathit{Instr} \cup \{handle\}$.

The control-flow graph *for a method* m *is defined as* $\mathtt{b}\mathcal{G}(m) = \bigcup_{\forall i_{pc} \in instr_m} \mathtt{b}\mathcal{G}(pc, i_{pc}, \bar{H}_m)$*, where* $instr_m$ *is the instruction array for method* m*, and* i_{pc} *is the instruction with array index pc. The* control-flow graph *for a closed program* p *is defined as* $\mathtt{b}\mathcal{G}(p) = \bigsqcup_{\forall m \in pc} \mathtt{b}\mathcal{G}(m)$.

The extraction rules work as follows. Assignments to a newly introduced temporary variables, denoted by the *TempAssign* set, do not produce edges. Such instructions are produced by the BIR transformation to keep data consistent, but they do not have a correspondent edge on the direct extraction, thus we can ignore them. For the instructions in *Assignment* set, `[nop]` and `[mayinit]` a normal transition to the next control node is generated. The conditional jump `[if` *expr* `pc']` produces a branch in the CFG: control can go either to the next control point, or to the branch point `pc'`. The unconditional jump `goto pc'` adds a single transition to control point `pc'`. The `[return]` and `[vreturn` *expr*`]` instructions generate an internal transition to a return node, i.e., a node with the atomic proposition r. Notice that, although both nodes are tagged with the same `pc`, they are different, because their sets of atomic propositions are different.

The extraction rule for calls to constructors (`[new C]`) produces a single normal edge, since there is only one possible receiver for the call. Also, we produce a pair of edges relatives to *NullPointerException*. The BIR transformation does not produce a correspondent [notnull] instruction for such case, and at first we should not support such exceptional flows. However the direct algorithm contemplates such case, thus we produce these two exceptional edges for the sake

of soundness. Moreover, $\bar{\mathcal{N}}_n^{pc}$ returns transitions to exceptional nodes due to uncaught exceptions, together with the appropriate exception handling transitions.

The extraction rule for method calls is similar to that of the direct extraction (in Section 3). Again, we assume that an appropriate virtual method calls resolution is used. We add a normal edge for each possible receiver returned from res_b^{α}. Again, $\bar{\mathcal{N}}_n^{pc}$ returns a pair of transitions for uncaught exceptions.

The `[throw` x`]` instruction, similarly to virtual method call resolution, depends on some kind of static analysis to find out the possible exceptions that can be thrown. The BIR transformation only provides the static type of the exception x . We define X as the set containing the static type of x and its subtypes. Thus we add one exceptional edge for each element of X, together with its correspondent edge after querying the exception table.

Finally, we cite the rule for assertion instructions. In this case, we create a normal edge, indicating that the execution was successful, one exceptional edge to mark the raise of an exception, a third edge, which shows if the instruction has an associated entry in the exceptions table.

4.4 CFG Extraction Correctness Proof

We now enunciate the correctness proof theorem for control-flow graphs extracted from the composition of `BC2BIR` and $b\mathcal{G}$ algorithms. We prove that given the same JBC program, the control-flow graph generated with the composition of algorithms simulates structurally the control-flow graph generated using the $m\mathcal{G}$ direct algorithm.

Theorem 2 (Structural Simulation of Control-Flow Graphs). *Let P be an arbitrary Java bytecode closed program. Then $b\mathcal{G} \circ$ `BC2BIR`(P) weakly simulates $m\mathcal{G}(P)$, considering the set $\mathcal{RE}$.*

The proof is stated using case analysis over the Java bytecode instructions set, and is available on-line [4]. Based on the previous proof that structural simulation implies behavioral simulation [9], we can conclude that the correctness of structural simulation of $m\mathcal{G}(P)$ by the control-flow graph produced in `BC2BIR`(P) implies also behavioural simulation.

5 Related Work

Sinha et. al. [13, 14] propose criteria for testing exception handling constructs in Java programs (Java source code). They consider the effect of exception propagation and exceptions type conversion. The proposed algorithm for CFG construction traverses the (Abstract Syntax Tree) AST of the program and then inter-procedural CFG (ICFG) is established. Normal CFG is constructed using algorithms proposed in [1].

[4] Available at `http://www.csc.kth.se/~pedrodcg/files/foveoos11-proof.pdf`

In a similar work Jiang [11] propose an algorithm to extract exceptional control-flow graph (ECFG) of C++ programs. In the proposed model of the programs implicit control-flow of exceptions and exceptions propagation is represented. Based on the inter-procedural ECFG (ICFG) they described techniques for path testing and definition-use testing of C++ programs.

Jo and Chang [12] propose a method to construct CFG by computing separately normal flow and exception flow of Java programs (Java source code). Using a set-constraints of exceptions and iterative fix-point method they compute exception propagation paths. They show that CFG of a program can be constructed by merging an exception flow graph onto a normal flow graph.

Our CFG extraction rules use the results of inter-procedural analysis and exception propagation from above mentioned work, however, none gives formal extraction rule and correctness proof.

6 Conclusion

This paper presents an efficient and precise control-flow graph extraction algorithm, and shows the proof outline of its correctness. To the best of our knowledge, this is the first control-flow graph extraction algorithm that has been proven correct. The proof is presented in pencil-and-paper style, but paves the ground for a second version using automated reasoning.

The algorithm is efficient and precise, because it uses an intermediate stackless representation. This allows to generate precise information about exceptional control-flow, and it keeps the generated control-flow graphs relatively small.

To prove correctness of the algorithm, i.e., to show that any behaviour of the extracted control-flow graph is an over-approximation of the program's behaviour, a second extraction algorithm is used that works directly on the bytecode. It is easy to prove correctness of this direct algorithm. To prove correctness of the indirect algorithm we show that the flow graphs it generates simulate structurally the flow graphs generated by the direct algorithm. Since structural simulation implies behavioural simulation, this gives us the desired result.

As future work, we are studying how the extraction algorithm could be adapted to a modular setting. Currently, only flow-graphs for complete programs can be extracted. However, our intention is to use the extracted flow-graphs as input for CVPP [10], a tool set for compositional verification of control-flow safety properties. In this setting, one often wishes to generate a flow-graph from an incomplete program. In addition, we are also studying how the techniques used in this paper can be used to prove correctness of an extraction algorithm that preserves some data of the original program, and how to use it for programs with multiple threads of execution.

Acknowledgments We would like to thank Dilian Gurov for his support and valuable comments. Also we thank the Celtique team in INRIA-Rennes for the cooperation and constant clarifications over the BIR language.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
2. Amighi, A.: Flow Graph Extraction for Modular Verification of Java Programs. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden (February 2011), http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/amighi_afshin_11038.pdf, Ref.: TRITA-CSC-E 2011:038
3. Barre, N., Demange, D., Hubert, L., Monfort, V., Pichardie, D.: Sawja api documenation (June 2011), http://javalib.gforge.inria.fr/doc/sawja-api/sawja-1.3-doc/api/index.html
4. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. J. of Computer Security 9(3), 217–250 (2001)
5. Demange, D., Jensen, T., Pichardie, D.: A provably correct stackless intermediate representation for java bytecode. Tech. Rep. 7021, Inria Rennes (2009), http://www.irisa.fr/celtique/demange/bir/rr7021-3.pdf, version 3, November 2010
6. Freund, S.N., Mitchell, J.C.: A type system for the java bytecode language and verifier. J. Autom. Reason. 30, 271–321 (August 2003), http://dx.doi.org/10.1023/A:1025011624925
7. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
8. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Formal Verification of Object–Oriented Software (FoVeOOS '10). LNCS, vol. 6528. Springer (2010)
9. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: International Conference on Formal Engineering Methods (ICFEM '08). LNCS, vol. 5256, pp. 147–166. Springer (2008)
10. Huisman, M., Gurov, D.: CVPP: A tool set for compositonal verification of control-flow safety properties. In: Formal Verification of Object–Oriented Software (FoVeOOS '10). LNCS, vol. 6528, pp. 107–121. Springer (2010)
11. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. SIGPLAN Not. 42, 3–8 (April 2007), http://doi.acm.org/10.1145/1288258.1288259
12. Jo, J.W., Chang, B.M.: Constructing control flow graph for java by decoupling exception flow from normal flow. In: ICCSA (1). pp. 106–113 (2004)
13. Sinha, S., Harrold, M.J.: Criteria for testing exception-handling constructs in java programs. In: Proceedings of the IEEE International Conference on Software Maintenance. pp. 265–. ICSM '99, IEEE Computer Society, Washington, DC, USA (1999), http://portal.acm.org/citation.cfm?id=519621.853364
14. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. 26, 849–871 (September 2000), http://portal.acm.org/citation.cfm?id=352825.352832