Provably Correct Control-Flow Graphs from Java Programs with Exceptions

Afshin Amighi[1], Pedro de C. Gomes[2], Dilian Gurov[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,dilian}@csc.kth.se

Abstract. We present an algorithm to extract flow graphs from Java bytecode, including exceptional control flows. We prove its correctness, meaning that the behavior of the extracted control-flow graph is a sound over-approximation of the behavior 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 various static analyses, such as model checking.
The extraction is performed in two phases. In the first phase the program is transformed into a BIR program, a stack-less intermediate representation of Java bytecode, from which the control-flow graph is extracted in the second phase. We use this intermediate format because it results in compact flow graphs, with provably correct exceptional control flow. To prove the correctness of the two-phase extraction, we also define an idealized extraction algorithm, whose correctness can be proven directly. Then we show that the behavior of the control-flow graph extracted via the intermediate representation is an over-approximation of the behavior of the directly extracted graphs, and thus of the original program. We implemented the indirect extraction as the CFGex tool and performed several test-cases to show the efficiency of the algorithm.

Keywords: Software Verification, Static Analysis, Program Models

1 Introduction

Over the last decade software has become omnipresent, and at the same time, 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 problem in this area is that the state space of software is enormous, often infinite. Therefore appropriate abstractions are necessary to make the formal analysis tractable. It is important that such abstractions 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,

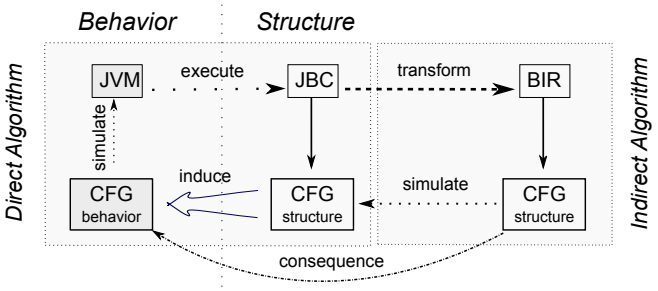

Fig. 1. Schema for CFG extraction and correctness proof

control-flow graphs (CFGs) [4] are a widely used abstraction, where only the control-flow information is kept, and all program data is abstracted away. Concretely, in a CFG, nodes represent the control points of a method, and edges represent the instructions that move between control points.

For two different reasons the analysis of exceptional flows is a major complication to soundly extract CFGs of Java bytecode. 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 decide to which handler (if any) control will be transferred. 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 care.

The literature contains several approaches to extract control-flow graphs automatically from program code. However, typically no formal argument is given to justify that the extraction is property-preserving. This paper fills this gap: it defines a flow graph extraction algorithm for Java bytecode (JBC), including exceptional control flow *and* it proves that the extraction algorithm is sound w.r.t. program behavior. 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. In particular, it includes explicitly thrown instructions, and a significant subset of run-time exceptions.

This paper defines two different extraction algorithms, where the first idealized algorithm is used to prove correctness of the second, which is implementable. This relationship is visualized in Figure 1.

The first extraction algorithm (in Section 3) creates flow graphs directly from Java bytecode. Its correctness proof is quite direct, but the resulting CFG 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 CFG. 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, verification of control flow properties on these CFGs is not so efficient, because of their size.

As an alternative, we also present a two-phase extraction algorithm using the Bytecode Intermediate Representation (BIR) language [7]. 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 more compact, which makes property verification more efficient. BIR has been developed by Demange *et al.* as a module of Sawja [10], 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. This allows us to have an efficient and provably correct extraction algorithm, including exceptional control flow. Our two-phase extraction algorithm first uses the transformation of Demange *et al* to generate BIR from JBC, and then extract CFGs from BIR. It is implemented as the tool CFGEx.

As mentioned above, the idealized direct extraction algorithm is used to prove correctness of the indirect extraction algorithm. This algorithm cannot be proven correct directly, because there is no behavior defined for BIR. Instead, we connect the BIR CFGs to the CFGs produced by the direct algorithm for the same program, and we show that every BIR CFG structurally simulates the JBC CFG. Then we use an existing result that structural simulation induces behavioral simulation (see [9]). In addition, we prove that the CFG produced by the direct algorithm behaviorally simulates the original Java bytecode program. From these two results we can conclude that all behaviors of the CFG generated by the indirect algorithm (BIR) are a sound over-approximation of the original program behavior. Thus, the extraction algorithm produces control-flow graphs that are sound for the verification of temporal safety properties.

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, proves its correctness, and presents experimental results. Section 5 presents the formal correntess argumentation, and the strutural simulation proof of CFGs. Finally Sections 6 and 7 present related work and conclude.

2 Preliminaries

This section briefly reviews a formalization of Java bytecode programs, their execution environment and a model for Java programs.

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 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. Programmers can also explicitly throw exceptions (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, execution terminates exceptionally.

We use Freund and Mitchell's formal framework for Java bytecode [8]. A JBC program is modeled as an environment Γ that is a partial map from class names, interface names and method signatures to their respective definitions. Sub-typing 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.

A JVM execution state is modeled as 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. The sequence is defined formally as:

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

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 that 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*.

```
public class Number{
 public static boolean even(int n) {
  try {
   if (n<0)
     throw new NegNumberException();
   else if (n==0)
     return true;
   } catch (NegNumberException e) { n *= -1;}
   return odd(n-1);
 }
 public static boolean odd(int n) {
   if(n==0)
     return false;
   else
     return even(n-1);
 }
}
```

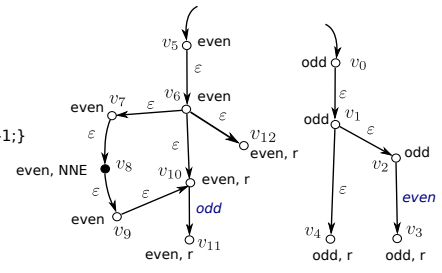

Fig. 2. Method specifications of methods `even` and `odd`

2.2 Program Model

Control-flow graphs are an abstract model of a program. To define the structure and behavior of a CFG we follow Gurov et al. and use the general notion of *model* [9, 11].

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,* $\rightarrow \subseteq S \times L \times S$ *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* $\mathcal{S}$ *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* specification with exceptions *for a method* $m \in \textsc{Meth}$ *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*,* $L_m = M \cup \{\varepsilon, handle\}$ *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.* $\mathbb{E}_m \subseteq V_M$ *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. We call edges labeled with ε silent transitions; the others are visible. Figure 2 shows a sample program with corresponding CFG.

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

Subset	Description	Samples
RetInst	Normal return instructions	`return`
CmpInst	Computational instructions	`nop, push v, pop`
CndInst	Conditional instructions	`ifeq q`
JmpInst	Jump instructions	`goto q`
XmpInst	Potentially can raise exceptions	`div, getfield f`
InvInst	Method invocations	`invokevirtual (o,m)`
ThrInst	Explicit exception throw	`throw X`

Fig. 3. Grouping of Bytecode instructions

Definition 3 (Flow Graph Interface). *A* flow graph interface *is a quadruple* $I = (I^+, I^-, E, M_e)$*, where* $I^+, I^- \subseteq$ Meth *are finite sets of provided and required method signatures,* $E \subseteq$ 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*.*

Now we define a method's flow graph as pair of its method's specification, and interface. A program's flow graph is the disjoint union of the flow graphs of all the methods defined in the program.

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

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

3 Extracting Control-Flow Graphs from Bytecode

This section describes how we build CFGs directly from the bytecode. The core of the algorithm is a set of rules that, given an instruction and address, produces a set of edges between the current control node, and all possible successors.

We group all JBC instructions into disjoint sets (Figure 3). In JBC, `athrow` does not have an argument; instead the exception is determined at run-time by the top of the stack. Static analysis of a JBC program can determine the possible types of the exceptions to be thrown by `athrow`. We use this to replace `athrow` with `throw X`, where `X` denotes the set of possible exception types.

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

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

The nodes of a method's CFG are defined as a mapping from the JVM configurations executing the method. All nodes are tagged with an address and a method signature. The set of addresses is extended by adding symbol $\flat$ 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, λ_m, and $\mathbb{E}_m$. For a node $v \in V_m$ indicating control point $\ell \in \text{ADDR}_\flat$ of method m, we define $v = (m, \ell)$. The labeling function λ_m specifies A_m for a given $v \in V_m$. If $m[\ell] \in \text{RETINST}$ then the node is tagged with r. If the node is an exceptional node then it is marked with the exception type $x \in E$. The 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: $v \vDash x$ means that node v is tagged with exception x; $\bullet_m^{\ell,x}$ indicates an exceptional control node and $\circ_m^\ell$ denotes a normal control node.

3.1 The Extraction Algorithm

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 5 (Method Control-Flow Graph Extraction). *Let V be the set of nodes and $L_m = M \cup \{\varepsilon, handle\}$, $M \subset \text{METH}$. Let Π be a set of environments. Then the control-flow graph of method m is extracted by $\mathtt{m}\mathcal{G} : \Pi \times \text{METH} \to \mathcal{P}(V \times L_m \times V)$, defined in Figure 4 (where succ denotes the next instruction address function).*

The construction rules are defined purely syntactically, based on the method's instructions. However, intuitively they are justified by 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 edges to the next control address and to the address specified for the jump (q) are generated. 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 $\mathcal{H}_p^x$. 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 a handler, it returns an edge from an exceptional node to a normal node. Otherwise it produces an edge 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 $\mathcal{X} : \text{XMPINST} \to \mathcal{P}(\text{EXCP})$ determines 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.

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

$$(\ \Gamma \vdash x <: y\ ) \implies \mathcal{H}_p^x = \begin{cases} \{(\bullet_m^{p,x}, handle, \circ_m^t)\} & \hbar_{\Gamma[m]}(p,y) = t \neq 0 \\ \{(\bullet_m^{p,x}, handle, \bullet_m^{p,x,r})\} & \hbar_{\Gamma[m]}(p,y) = 0 \ \wedge\ m \notin M_e \\ \{(\bullet_m^{p,x}, handle, \bullet_m^{\flat,x,r})\} & \hbar_{\Gamma[m]}(p,y) = 0 \ \wedge\ m \in M_e \end{cases}$$

$$\mathcal{N}_p^i = \left\{\{(\circ_m^p, handle, \bullet_m^{p,x})\} \cup \mathcal{H}_p^x |\ \bullet_n^{q,x,r} \in nodes(\mathtt{m}\mathcal{G}(n)),\ n \in Rec_\Gamma(i)\right\}$$

$$\mathcal{R}_p^i = \{(\circ_m^p, n, \circ_m^{succ(p)}) \mid\ n \in Rec_\Gamma(i)\}$$

$$\mathcal{E}_p^i = \left\{\{(\circ_m^p, \varepsilon, \bullet_m^{p,x})\} \cup \mathcal{H}_p^x \mid x \in \mathcal{X}(i)\right\}$$

$$\mathtt{m}\mathcal{G}(S_1; S_2, H) = \mathtt{m}\mathcal{G}(S_1, H) \cup \mathtt{m}\mathcal{G}(S_2, H)$$

$$\mathtt{m}\mathcal{G}((p,i),H) = \begin{cases} \{(\circ_m^p, \varepsilon, \circ_m^{succ(p)})\} & \texttt{if}\ i \in \textsc{CmpInst} \\ \{(\circ_m^p, \varepsilon, \circ_m^q)\} & \texttt{if}\ i \in \textsc{JmpInst} \\ \{(\circ_m^p, \varepsilon, \circ_m^{succ(p)}), (\circ_m^p, \varepsilon, \circ_m^q)\} & \texttt{if}\ i \in \textsc{CndInst} \\ \left\{\{(\circ_m^p, \varepsilon, \bullet_m^{p,x})\} \cup \mathcal{H}_p^x \mid x \in X\right\} & \texttt{if}\ i = \texttt{throw}\ X \\ \{(\circ_m^p, \varepsilon, \circ_m^{succ(p)})\} \cup \mathcal{E}_p^i & \texttt{if}\ i \in \textsc{XmpInst} \\ \{(\circ_m^p, \varepsilon, \bullet_m^{p,\varrho_N})\} \cup \mathcal{H}_p^{\varrho_N} \cup \mathcal{R}_p^i \cup \mathcal{N}_p^i & \texttt{if}\ i \in \textsc{InvInst} \end{cases}$$

Fig. 4. CFG Construction Rules

To extract edges for method invocations, function $Rec_\Gamma(i)$ determines the set of possible method signatures of a method call in environment Γ. Index of the signature shows the type of the receiver object in method call. The receiver object for `invokevirtual` is determined by late binding. The virtual method call resolution function res_Γ^α will be used, where α is a standard static analysis technique to resolve the call.

$$Rec_\Gamma(i) = \begin{cases} \{n_{staticT(o)}\} \text{ if } i \in \{\texttt{invokespecial}\ (\text{o,n}), \texttt{invokestatic}\ (\text{o,n})\} \\ \{n_\tau \mid \tau \in res_\Gamma^\alpha(o,n)\} \text{ if } i = \texttt{invokevirtual}\ (\text{o,n}) \end{cases}$$

For example, Rapid Type Analysis (`RTA`) [2] returns the set of subtypes of the callee's static type which are instantiated at some part of the program. I.e. created by a `new` instruction. If the *RTA* algorithm, i.e., $\alpha = RTA$, then the result of the resolution for object o and method n in environment Γ will be:

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

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.

$\mathcal{R}_p^i$ is the set of the edges for normally terminating calls, $\mathcal{H}_p^{\varrho_N}$ is the set of edges to handle ϱ_N, and $\mathcal{N}_p^i$ defines the set of edges to handle all uncaught exceptions from all possible callees. We add the callee's signature as an index to 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, $\mathcal{H}_p^x$ generates edges for successful/failed handlers for all exceptional nodes in CFG_n i.e., the CFG of method $n \in res_\Gamma^\alpha(o,n)$.

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

3.2 Correctness of $\mathtt{m}\mathcal{G}$

To prove soundness of the flow graph extraction, we have to define the behavior of flow graphs. The following extends the behavior definition of flow graphs from [11], based on our extraction rules.

Definition 6 (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 $\lambda_g((v,\sigma)) = \lambda(v)$*
- *$\to_g \subset S_g \times S_g$ 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* $m \in I^+,\ v \xrightarrow{\varepsilon}_m v', v \models \neg r,\ v \not\models \textsc{Excp},\ v' \not\models \textsc{Excp}$

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

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

To show correctness of the extraction algorithm, we show that the extracted CFG of method m can match all possible moves during the execution of m. We first define a mapping θ that abstracts JVM configurations to CFG behavioral configurations, and we use this to prove that the behavior of a CFG simulates the behavior of the corresponding method in JBC.

Definition 7 (Abstraction Function for VM States). *Let* VJVM *be the set of JVM execution configurations and S_g the set of states in $\mathtt{m}\mathcal{G}$. Then θ : VJVM $\rightarrow S_g$ is defined inductively as follows:*

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

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 INST $\cup \{\epsilon\}$ to the CFG transition labels in L_m. When an exception happens, JVM takes the control of the execution to handle the exception. There is no instruction in JBC instructions set to accomplish handling. We call these transitions as silent transitions and label them with ϵ.

Now we enunciate the Theorem 1, which states the behavioral simulation of JVM. 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. We show that the set S corresponds to the configurations $\Theta(C)$. To show that this indeed holds, we use a case analysis on VJVM. For the complete proof, we refer to Amighi's Master thesis [1].

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

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 `BC2BIR` transformation from JBC to BIR. Then, we discuss how BIR is transformed into CFGs, enunciate the correctness proof, and discuss practical results.

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. We give a brief overview of BIR, for a full account we refer to [7].

Figure 5 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

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

$$
\begin{array}{rcl}
\mathit{Assignment} & ::= & \mathit{target}\ \texttt{:=}\ \mathit{expr} \\
\mathit{Return} & ::= & \texttt{vreturn}\ \mathit{expr} \mid \texttt{return} \\
\mathit{MethodCall} & ::= & \mathit{expr}.\texttt{ns(}\mathit{expr}\texttt{,...,}\ \mathit{expr}\texttt{)} \\
& & \mid \mathit{target} := \mathit{expr}.\texttt{ns(}\mathit{expr}\texttt{,...,}\mathit{expr}\texttt{)} \\
\mathit{NewObject} & ::= & \mathit{target} := \texttt{new C(}\mathit{expr}\texttt{,...,}\mathit{expr}\texttt{)} \\
\mathit{Assertion} & ::= & \texttt{notnull}\ \mathit{expr} \mid \texttt{notzero}\ \mathit{expr}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{instr} & ::= & \texttt{nop} \mid \texttt{if}\ \mathit{expr}\ \texttt{pc} \mid \texttt{goto pc} \\
& & \mid \texttt{throw}\ \mathit{expr} \mid \texttt{mayinit C} \\
& & \mid \mathit{Assignment} \mid \mathit{Return} \\
& & \mid \mathit{MethodCall} \mid \mathit{NewObject} \\
& & \mid \mathit{Assertion}
\end{array}
$$

Fig. 5. Expressions and Instructions of BIR

Exception	*Assertion*	*Exception*	*Assertion*
NullPointerException	`[notnull]`	*ArithmeticException*	`[notzero]`
IndexOutOfBoundsException	`[checkbound]`	*ClassCastException*	`[checkcast]`
NegativeArraySizeException	`[notneg]`	*ArrayStoreException*	`[checkstore]`

Fig. 6. Implicit exceptions supported by BIR, and associated assertions

and booleans; these are transformed into methods calls by the `BC2BIR` transformation. It also 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.

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

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, similarly to the `athrow` instruction in JBC. Method call instructions are represented by their method signature. For non-*void* methods, the instruction assigns the result value to a variable.

In contrast to JBC, object allocation and initialization happen in a single step, during the execution of the `new` instruction. However, Java also has class initialization, i.e., the one-time initialization of a class's static fields. 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.

BIR models implicit exceptions by inserting special assertions before the instructions that can potentially raise an exception, as defined for the JVM. Figure 6 shows all implicit exceptions that are currently supported by the `BC2BIR` transformation [3], and the associated assertion. For example, the transforma-

tion inserts a `[notnull]` assertion before any instruction that might throw a *NullPointerException*, such as an access to a reference. If the assertion holds, it behaves as a `[nop]`, and control-flow passes to the next instruction. If the assertion fails, control-flow is passed to the exception handling mechanism. In the transformation from BIR to CFG, we use a function $\bar{\chi}$ to obtain the exception associated with an instruction. Notice that our translation from BIR to CFG can easily be adapted for other implicit exceptions, provided appropriate assertions are generated for them.

A BIR program is organized in exactly the same way as a Java bytecode program. A program is a set of classes, ordered by a class hierarchy. Every class consists of a name, methods and fields. A method's code is stored in an instruction array. 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. It translates a complete JBC program into BIR by symbolically executing the bytecode using an abstract stack. This stack is used to reconstruct expression trees and to connect instructions to its operands. As we are only interested in the set of BIR instruction that can be produced, we do not discuss all details of this transformation. For the complete algorithm, we refer to [7].

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 function returns the $Fail$ element, and the `BC2BIR` algorithm aborts. The function $\texttt{BC2BIR}_{instr}$ is defined as follows.

Definition 8 (BIR Transformation Function). *Let* $AbsStack \in Expr*$. *The rules defining the* instruction-wise transformation $\texttt{BC2BIR}_{instr} : \mathbb{N} \times instr_{JBC} \times AbsStack \rightarrow (instr_{BIR}* \times AbsStack) \cup Fail$ *from Java bytecode into BIR are given in Figure 7.*

As a convention, we use brackets to distinguish BIR instructions from their JBC counterpart. Variables $\texttt{t}^{\texttt{i}}_{\texttt{pc}}$ are new, introduced by the transformation.

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

The `store x` instruction can produce one or two assignments, depending on the state of the abstract stack. The `putfield f` outputs a set of BIR instructions: `[notnull` e`]` guards if the e is a valid reference; then the auxiliary function $FSave$ introduces a set of assignment instructions to temporary variables; and finally the assignment to the field (`e.f`) is generated. Similarly,

Input	Output
`pop`	$\emptyset$
`push c`	$\emptyset$
`dup`	$\emptyset$
`load x`	$\emptyset$
`add`	$\emptyset$

Input	Output
`nop`	`[nop]`
`if` p	`[if e pc']`
`goto` p	`[goto pc']`
`return`	`[return]`
`vreturn`	`[return e]`

Input	Output
`div`	`[notzero` $\mathtt{e}_2$`]`
`athrow`	`[throw e]`
`new C`	`[mayinit C]`
`getfield f`	`[notnull e]`

Input	Output
`store x`	`[x:=e]` or `[`$\mathtt{t}^0_{\mathtt{pc}}$`:=x;x:=e]`
`putfield f`	`[notnull e;`*FSave*`(pc,f,`*as*`);e.f:=e'` `]`
`invokevirtual m`	`[notnull e;`*HSave*`(pc,`*as*`);`$\mathtt{t}^0_{\mathtt{pc}}$`:=e.ns(`$\mathtt{e}'_1 \ldots \mathtt{e}'_n$`)]`
`invokespecial m`	`[notnull e;`*HSave*`(pc,`*as*`);`$\mathtt{t}^0_{\mathtt{pc}}$`:=e.ns(`$\mathtt{e}'_1 \ldots \mathtt{e}'_n$`)]` or `[`*HSave*`(pc,`*as*`);`$\mathtt{t}^0_{\mathtt{pc}}$`:=new C(`$\mathtt{e}'_1 \ldots \mathtt{e}'_n$`)]`

Fig. 7. Rules for $\mathtt{BC2BIR}_{instr}$

Java bytecode	BIR
0: `iload_0`	
1: `ifne 6`	0: `if ($bcvar0 != 0) goto 2`
4: `iconst_0`	
5: `ireturn`	1: `vreturn 0`
6: `aload_0`	
7: `iconst_1`	
8: `isub`	2: `mayinit Number`
9: `invokestatic Number.even(int)`	3: `$irvar0 := Number.even($bcvar0 - 1)`
12: `ireturn`	4: `vreturn $irvar0`

Fig. 8. Comparison between instructions in method odd()

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

Figure 8 shows the JBC and BIR versions of method `odd()` (from Figure 2). The different colors show the collapsing of instruction by the transformation. The BIR method has a local variable (`$bcvar0`) and a newly introduced variable (`$irvar0`). We observe reconstructed expression trees as the argument to the method invocation, and as the operand to the `[if]` instruction. The `[mayinit]` instruction shows that class `Number` can be initialized on that program point.

4.3 Transformation from BIR into Control-Flow Graphs

The extraction algorithm that generates a CFG from BIR iterates over the instructions of a method, using the transformation function $\mathtt{b}\mathcal{G}$, that takes as input

a program counter and an instruction array for a BIR method. Each iteration outputs a set of edges.

To define $\mathtt{b}\mathcal{G}$, we introduce several auxiliary functions and definitions similar to the ones introduced for the direct extraction (in Section 3). As a convention, we use bars (e.g., $\bar{N}$) to differentiate the similar functions from the direct, and indirect algorithms.

First, $\bar{H}$ is the exception table for a given method, containing the same entries as the JBC table, but its control points relate to BIR instructions. The function $\hbar_{\bar{H}}(\mathtt{pc}, x)$ searches for the first handler for the exception x (or a subtype) at position pc. given a virtual method call resolution algorithm α. The function $\bar{\mathcal{H}}_x^{\mathtt{pc}}$ returns an edge after querying $\hbar$ for exception handlers. The function $\bar{\mathcal{N}}_n^{\mathtt{pc}}$ adds exceptional edges, relative to exceptions propagated by the called method. Its computation requires the previous extraction of CFGs from the called method.

The extraction is parametrized by a virtual method call resolution algorithm α. The function $res^{\alpha}(\mathtt{ns})$ uses α to return a safe over-approximation of the possible receivers to a virtual invocation of a method with signature ns, or the single receiver if a call is non-virtual (e.g., to a static method).

We divide the definition of $\mathtt{b}\mathcal{G}$ into two parts. The *intra-procedural* analysis extracts a CFG for every method, based solely on its instruction array, and its exception table. The *inter-procedural* analysis computes $\bar{\mathcal{N}}_n^{\mathtt{pc}}$, the set of edges that can follow a method call, and potential exception propagation.

Definition 9 (Control Flow Graph Extraction). *The* control-flow graph extraction function $\mathtt{b}\mathcal{G} : (Instr \times \mathbb{N}) \times \bar{H} \to \mathcal{P}(V \times L_m \times V)$ *is defined by the rules in Figure 9. Given method* m*, with* $ArInstr_m$ *as its instruction array, the* control-flow graph *for* m *is defined as* $\mathtt{b}\mathcal{G}(m) = \bigcup_{i_{\mathtt{pc}} \in ArInstr_m} \mathtt{b}\mathcal{G}(i_{\mathtt{pc}}, \bar{H}_m)$*, where* $i_{\mathtt{pc}}$ *denotes the instruction with array index* pc*. Given a closed BIR program* Γ_B*, its* control-flow graph *is* $\mathtt{b}\mathcal{G}(\Gamma_B) = \bigcup_{m \in \Gamma_B} \mathtt{b}\mathcal{G}(m)$*.*

First, we describe the rules applied by the intra-procedural analysis. Instructions that store expressions (i.e., *assignments*), `[nop]` and `[mayinit]` add a single edge to the next normal control node. 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 edge to control point `pc'`. The `[return]` and `[vreturn` *expr*`]` instructions generate an internal edge to a return node, i.e., a node with the atomic proposition r. Notice that, although both nodes are tagged with the same pc, they are different, because their sets of atomic propositions are different.

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

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

The `[throw` x`]` instruction, similarly to virtual method call resolution, depends on a static analysis to find out the possible exceptions that can be thrown.

$$\bar{\mathcal{H}}^{\mathtt{pc}}_x = \begin{cases} \{\ (\bullet^{\mathtt{pc},x}_m, \mathit{handle}, \circ^{\mathtt{pc'}}_m)\ \} & \text{if } \hbar_{\bar{H}}(\mathtt{pc}, x) = \mathtt{pc'} \neq 0 \\ \{\ (\bullet^{\mathtt{pc},x}_m, \mathit{handle}, \bullet^{\mathtt{pc},x,r}_m)\ \} & \text{if } \hbar_{\bar{H}}(\mathtt{pc}, x) = 0 \end{cases}$$

$$\mathtt{b}\mathcal{G}(i_{\mathtt{pc}}, \bar{H}) = \begin{cases} \{(\circ^{\mathtt{pc}}_m, \varepsilon, \circ^{\mathtt{pc+1}}_m)\} & \text{if } i \in \mathit{Assignment} \cup \{\mathtt{[nop]}, \mathtt{[mayinit]}\} \\ \{(\circ^{\mathtt{pc}}_m, \varepsilon, \circ^{\mathtt{pc+1}}_m), (\circ^{\mathtt{pc}}_m, \varepsilon, \circ^{\mathtt{pc'}}_m)\} & \text{if } i = \mathtt{[if}\ \mathit{expr}\ \mathtt{pc']} \\ \{(\circ^{\mathtt{pc}}_m, \varepsilon, \circ^{\mathtt{pc'}}_m)\} & \text{if } i = \mathtt{[goto\ pc']} \\ \{(\circ^{\mathtt{pc}}_m, \varepsilon, \circ^{\mathtt{pc},r}_m)\} & \text{if } i \in \mathit{Return} \\ \{(\circ^{\mathtt{pc}}_m, \mathtt{C}, \circ^{\mathtt{pc+1}}_m), (\circ^{\mathtt{pc}}_m, \varepsilon, \bullet^{\mathtt{pc},\varrho N}_m)\} \cup \bar{\mathcal{H}}^{\mathtt{pc}}_{\varrho N} \cup \bar{\mathcal{N}}^{\mathtt{pc}}_{\mathtt{C}} & \text{if } i \in \mathit{NewObject} \\ \bigcup_{n \in res^{\alpha}(\mathtt{ns})} \{(\circ^{\mathtt{pc}}_m, n, \circ^{\mathtt{pc+1}}_m)\} \cup \bar{\mathcal{N}}^{\mathtt{pc}}_n & \text{if } i \in \mathit{MethodCall} \\ \bigcup_{x \in X}\ \{(\circ^{\mathtt{pc}}_m, \varepsilon, \bullet^{\mathtt{pc},x}_m)\} \cup \bar{\mathcal{H}}^{\mathtt{pc}}_x & \text{if } i = \mathtt{[throw}\ X\mathtt{]} \\ \{(\circ^{\mathtt{pc}}_m, \varepsilon, \circ^{\mathtt{pc+1}}_m), (\circ^{\mathtt{pc}}_m, \varepsilon, \bullet^{\mathtt{pc},\bar{\chi}(i)}_m)\} \cup \bar{\mathcal{H}}^{\mathtt{pc}}_{\bar{\chi}(i)} & \text{if } i \in \mathit{Assertion} \end{cases}$$

$$\bar{\mathcal{N}}^{\mathtt{pc}}_n = \bigcup_{\bullet^{\mathtt{pc},x,r}_n \in \mathtt{b}\mathcal{G}(n)} \{(\circ^{\mathtt{pc}}_m, \mathit{handle}, \bullet^{\mathtt{pc},x}_m)\} \cup \bar{\mathcal{H}}^{\mathtt{pc}}_x$$

Fig. 9. Extraction rules for control-flow graphs from BIR

The BIR transformation only provides the static type of the exception x. Let X be the set containing the static type of x and its subtypes. The transformation produces an exceptional edge for each element of X, followed by the appropriate edge derived from the exception table.

Finally, for each assertion, we produce a normal edge, and an exceptional edge, together with the appropriate edge derived from the exception table.

Next, we describe the inter-procedural analysis. In all program points where there is a method invocation, the function $\bar{\mathcal{N}}^{\mathtt{pc}}_n$ adds exceptional edges, relative to propagated exceptions by called methods. It analyzes if the CFG of an invoked method n contains an exceptional return node. If it does, then function $\bar{\mathcal{H}}^{\mathtt{pc}}_x$ verifies whether the exception x is caught in position `pc`. If so, it adds an edge to the handler. Otherwise it adds an edge to an exceptional return node.

In the later case, the propagation of the exception continues until it is caught by some caller method, or there are no more methods to handle it. This process can be performed using a fix-point computation.

4.4 Implementation

The extraction rules from Figure 9 are implemented in our CFG extraction tool CFGEx. It uses Sawja for virtual method call resolution (using RTA) and for the transformation from Bytecode into BIR. Table 1 provides statistics about the CFG extraction for several examples. All experiments are done on a server with an Intel i5 2.53 GHz processor and 4GB of RAM. Methods from the API are not extracted; only classes that are part of the program are considered.

BIR Time is the time used for Sawja to transform JBC into BIR. We divided the extraction of CFGs from BIR into two stages. First, the *intra-procedural* analysis extracts control-flow graphs for each BIR method by applying the formal rules in Figure 9, except the function $\bar{\mathcal{N}}$. As described in Section 4.3, this is

Software	# of JBC instr.	# of BIR instr.	BIR time (ms)	Intra-Procedural			Inter-Procedural		
				# of nodes	# of edges	time (ms)	# of nodes	# of edges	time (ms)
Jasmin	30930	10850	267	19152	19460	320	21651	21966	25
JFlex	53426	20414	706	38240	38826	859	42442	43072	23
Groove Ima.	193937	77620	587	159046	158593	4817	193268	192905	1849
Groove Gen.	328001	128730	926	251762	252102	13609	308164	308638	5541
Groove Sim.	427845	167882	1072	311008	311836	16067	386553	387556	6886
Soot	1345574	516404	98692	977946	976212	264690	1209823	1208358	57621

Table 1. Statistics for CFGEX

computed by an *inter-procedural* analysis. It extracts the transitions related to exceptions that are propagated from called methods. We compute this information using the fix-point algorithm of Jo and Chang [14].

Table 1 shows that the number of BIR instructions is less than 40 % of Bytecode instructions, for all cases. This indicates that the use of BIR avoids the blow-up of flow-graphs, and clearly program analysis benefits from this. We can also see that, on average, the computation time for intra and inter-procedural analysis grows proportionally with the number of BIR instructions. However, this growth depends heavily on the number of exceptional paths in the analyzed program.

5 Correctness of $\mathtt{b}\mathcal{G} \circ \mathtt{BC2BIR}$

We introduce the necessary notions and notations before stating the correctness proof. First, we define the notion of a well-formed Java bytecode program. Informally, such programs are the ones that are successfully loaded and start to execute by the Java Virtual Machine (JVM). In addition to being solely interested in programs that can actually be executed, we also use the hypothesis of well-formation to state the proof. E.g., the JVM will not start the execution of a program which contains a method that can terminate by running out of instructions, and not by reaching a return instruction.

Definition 10 (Well-Formed Java Program). *A well-formed Java bytecode program is a closed program which passes the JVM bytecode verification* [4].

Next we present the notion of weak transition relation for models, which follows the standard definition from Milner [15]. As usual, we write $p_i \xrightarrow{l} p_j$ to denote $(p_i, l, p_j) \in \rightarrow$, for some relation $\rightarrow$. Also, we use the ε label to denote silent transitions.

Definition 11 (Weak transition relation). *Given an arbitrary model* $(S, L \cup \{\varepsilon\}, \rightarrow, A, \lambda)$*, the relations* $\Longrightarrow \subseteq S \times S$*,* $\overset{\beta}{\Longrightarrow} \subseteq S \times L \times S$ *are defined as follows:*

[4] Requirements available at `http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html`

1. $p_i \implies p_j$ *means that there is a sequence of zero or more silent transitions from* p_i *to* p_j*. Formally,* $\implies \stackrel{def}{=} \stackrel{\varepsilon}{\rightarrow}^*$*, the transitive reflexive closure of* $\stackrel{\varepsilon}{\rightarrow}$*.*
2. $p_i \stackrel{\beta}{\implies} p_j$ *means that there is a sequence containing a single visible transition labeled with* β*, and zero or more silent transitions. Formally,* $\stackrel{\beta}{\implies} \stackrel{def}{=} \implies \stackrel{\beta}{\rightarrow} \implies$*.*

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

Definition 12 (Weak Simulation over Method Specifications). *Let* $\mathcal{M}_p = (S_p, L_p, \rightarrow_p, A_p, \lambda_p) : E_p$ *and* $\mathcal{M}_q = (S_q, L_q, \rightarrow_q, A_q, \lambda_q) : E_q$ *be two method specifications, and* $R \subseteq S_p \times S_q$*. Then* R *is a weak simulation if for all* $(p_i, q_j) \in R$ *the following holds:*

1. $\lambda_p(p_i) = \lambda_q(q_i)$
2. *if* $p_i \stackrel{\beta}{\implies} p_j$ *then there is* $q_j \in S_q$ *such that* $q_i \stackrel{\beta}{\implies} q_j$
3. $(p_j, q_j) \in R$*.*

We say that q *(weakly) simulates* p *if* $(p, q) \in R$*, for some weak simulation relation* R*. Also we say that* $\mathcal{M}_q$ *(weakly) simulates* $\mathcal{M}_p$ *if for all* $p \in E_p$*, there is* $q \in E_q$ *such that* q *(weakly) simulates* p*.*

The following proposition is a consequence of Definition 12, also presented in the standard definition by Milner. To prove weak simulation, it suffices to show that for every edge produced by the direct algorithm ("strong" transition), there is a matching weak transition with the same label produced by the indirect algorithm.

Proposition 1. *A relation* R *is a weak simulation if and only if for all* $(p_i, q_i) \in R$*, the following holds:*

1. *if* $p_i \stackrel{\varepsilon}{\rightarrow} p_j$ *then there is* q_j *such that* $q_i \implies q_j$ *and* $(p_j, q_j) \in R$*.*
2. *if* $p_i \stackrel{\beta}{\rightarrow} p_j$ *then there is* q_j *such that* $q_i \stackrel{\beta}{\implies} q_j$ *and* $(p_j, q_j) \in R$*.*

The flow graph from a program is the disjoint union of the flow graphs of all its methods, called method specifications. Thus it suffices to state the proof over an arbitrary method m, since the proof can be generalized to all methods, consequently to the entire program.

Along the text, we have used an informal definition of CFG nodes, which was sufficient for the understating, to avoid the overload of definitions. Now we present formally the notion of nodes in the CFG from a Java program. The definition of nodes for BIR programs is analogous, but uses `pc` to denote a position in the instructions array.

Definition 13 (Control Flow-Graph Nodes). *The set of nodes in a control-flow graph is defined as* $V \subseteq \textsc{Meth} \times \mathbb{N} \times \{e \in \mathcal{P}(\textsc{Excp}), |e| \leq 1\} \times \{\ \{\},\ \{r\}\ \}$*.*

The Definition 13 says that nodes from a flow graph are uniquely identified by a method signature, its position in the method's instruction array, a set containing the "return" atomic proposition, or empty; and a set containing a single exception, or empty. Once again, we use the notation $\circ_m^{p,x,y} and \bullet_m^{p,x,y}$, being the former used to stress when $x = \{\}$, and the latter used when the set x contains an exception. If the set $y = \{\}$ we may omit it, but if it is not, we add the r.

Finally, the `BC2BIR` transformation may collapse many bytecode instructions into one or more BIR instructions. This mapping of many-to-many instructions makes the proof statement cumbersome. Thus, we present the proof outline to help the reader to understand the nuances of the actual proof.

Proof Outline We divide the bytecode instructions into two sets: the *relevant* instructions are those that produce at least one BIR instruction in function $\texttt{BC2BIR}_{instr}$; the *irrelevant* instructions are those that produce none. Following Figure 7, `store` and `invokevirtual` are examples of relevant instructions; `add` and `push` are examples of irrelevant ones.

Next, we define *bytecode segments* as partitions over the array of bytecode instructions, delimited by each relevant instruction. Thus a bytecode segment contains zero or more irrelevant contiguous instructions, followed by a single relevant instruction. Such partitioning has to exist because of the Definition 10. It guarantees that well-formed bytecode programs must terminate after executing a `return` instruction, or a `throw` instruction that can not be caught. Both `return` and `throw` are relevant instructions. Thus, there can not be set of contiguous instructions which are not delimited by a relevant instruction.

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

Figure 8 illustrates the partitioning of instructions. The method `odd` contains four bytecode segments, and its corresponding BIR segments, grouped by colors. The relevant instructions are underlined.

We now explain the impact of the irrelevant instructions into the sub-graph of its segment. The set of irrelevant instructions is defined by all bytecode instructions that do not produce BIR instructions in the function $\texttt{BC2BIR}_{instr}$. The Definition 8 gives these instructions, which are `pop`, `push`, `dup`, `load` and `add`.

Figure 3 shows that all those instructions belong to the subset CmpInst of normal computation instructions. Moreover, the Definition 4 presents the following extraction rule for all the instructions $i \in$ CmpInst:

$$\mathtt{m}\mathcal{G}((p,i),H) = \{\ \circ_m^p \xrightarrow{\varepsilon} \circ_m^{\texttt{succ(p)}}\ \}$$

This means that irrelevant instructions produce a transition from the node tagged with control point p to the node tagged with the next position in the bytecode array.

Bytecode segments are defined as sequences of zero-to-many contiguous irrelevant instructions in the instruction array, followed by a single relevant instruction. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path with the same size as the number of irrelevant instructions.

Below we illustrate the pattern for the path graph, being $\mathtt{i}$ the position for the first irrelevant instruction, and $\mathtt{p}$ the position of the relevant instruction. In case the number of irrelevant instructions is zero, then $\mathtt{i} = \mathtt{p}$.

$$\circ_m^{\mathtt{i}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ(i)}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ(succ(i))}} \xrightarrow{\varepsilon} \ldots \circ_m^{\mathtt{p}} \ldots$$

Now we show that for each bytecode segment in a given method, the sub-graph produced by the direct algorithm is weakly simulated by the sub-graph produced transforming this segment into BIR instructions, and then extracting the control-flow graph with the $\mathtt{b}\mathcal{G}$ function.

Theorem 2 (Structural Simulation of Method Specifications). *Let Γ be a well-formed Java bytecode program, $\Gamma[m]$ the implementation of method with signature m, $\mathcal{RE}$ the subset of implicit exceptions that instructions can raise. Then $(\mathtt{b}\mathcal{G} \circ \mathtt{BC2BIR})(\Gamma[m])$ weakly simulates $\mathtt{m}\mathcal{G}(\Gamma[m])$, i.e, the method graph extracted using the indirect algorithm weakly simulates the method graph using the direct algorithm.*

Proof. We define a binary relation R as follows:

$$R \stackrel{def}{=} \{(\circ_m^{p,x,y}, \circ_m^{\mathtt{pc},x,y}) | seg_{jbc}(m,p) = seg_{bir}(m,\mathtt{pc}) \ \wedge \ \mathtt{pc} = min(seg_{bir}(m,\mathtt{pc}),x,y)\}$$

where $\circ_m^{p,x,y}$ is a control node in $\mathtt{m}\mathcal{G}(\Gamma[m])$ and $\circ_m^{\mathtt{pc},x,y}$ is a control node in $(\mathtt{b}\mathcal{G} \circ \mathtt{BC2BIR})(\Gamma[m])$. We introduce the auxiliary functions seg_{jbc} and seg_{bir}, which receive a position in the JBC and BIR instruction arrays, respectively, and return the index of the associated segment; and min returns the smallest $\mathtt{pc}$ among nodes from the same segment with the same sets x and y.

We stress the use of p for an arbitrary index in the bytecode array, and $\mathtt{pc}$ for an index in the BIR instructions array. During the proof we omit the use of abstract stacks, since only the instructions are relevant to produce the transitions. Also, we use the term "simulates", instead of "weakly simulates", for brevity.

Proposition 1 is used to show that R is a weak simulation. We relate the entry nodes in the sub-graphs produced by both algorithms. Then we show that, for all bytecode segments, the sub-graph produced by the indirect algorithm weakly simulates the sub-graph produced by the direct algorithm. The sub-graphs compose since we show that all the sink nodes are either return nodes (tagged with $y = \{r\}$), thus have no successors; or are normal nodes ($x = \{\}$), thus are entry nodes for sub-graphs of other segment.

Let $(\circ_m^{p,x,y}, \circ_m^{\mathtt{pc},x,y})$ be an arbitrary pair in R. The proof proceeds by case analysis on the type of the relevant instruction of the bytecode segment $seg_{jbc}(m,p)$.

We present the cases using the subsets JBC instructions presented in Figure 3. Those subsets group the instructions which share the same extraction rule in the direct algorithm.

Case relevant instruction $i \in$ CmpInst:

There are two relevant instruction in this subset: `nop` and `store`. The direct extraction produces a single transition from one normal node to the node tagged with the successor of p in the instructions array:

$$\mathtt{m}\mathcal{G}((p,i),H) = \{\ \circ_m^p \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ}(p)}\ \}$$

First, we analyze the indirect algorithm for the `store` instruction. The transformation $\mathtt{BC2BIR}_{instr}$ can return either one or two assignments. Applying the extraction rules $\mathtt{b}\mathcal{G}$ function, which have:

$$\mathtt{BC2BIR}_{instr}(p,\mathtt{store}) = \begin{cases} \mathtt{[x:=e]} & (\text{Case } I) \\ \mathtt{[t^0_{pc}:=x]\,;[x:=e]} & (\text{Case } II) \end{cases}$$

$$\mathtt{b}\mathcal{G}(\mathtt{[x:=e]_{pc}},\bar{H}) = \{\circ_m^{\mathtt{pc}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc+1}}\} \qquad (\text{Case } I)$$

$$\mathtt{b}\mathcal{G}(\mathtt{[t^0_{pc}:=x]_{pc}},\bar{H}) = \{\circ_m^{\mathtt{pc}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc+1}}\} \qquad (\text{Case } II)$$
$$\mathtt{b}\mathcal{G}(\mathtt{[x:=e]_{pc}},\bar{H}) = \{\circ_m^{\mathtt{pc+1}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc+2}}\}$$

For all nodes $\circ_m^i$ in the path graph created by the irrelevant instructions, we have that $(\circ_m^i, \circ_m^{\mathtt{pc}}) \in R$ since for all $\circ_m^i \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ}(i)}$ exists $\circ_m^{\mathtt{pc}} \implies \circ_m^{\mathtt{pc}}$ (because of reflexivity of $\implies$). Thus also $(\circ_m^{\mathtt{succ}(i)}, \circ_m^{\mathtt{pc}}) \in R$. The path graph will terminate on the node $\circ_m^{\mathtt{succ}(i)} = \circ_m^p$, which is the one tagged with the position p from the relevant instruction. This fact will be reused along the proof to explain the nodes produced by irrelevant instructions.

In the case where `store` produces a single assignment, we have that $(\circ_m^p, \circ_m^{\mathtt{pc}}) \in R$. There exists the transition $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ}(p)}$, and there is also $\circ_m^{\mathtt{pc}} \implies \circ_m^{\mathtt{pc+1}}$, Thus also $(\circ_m^{\mathtt{succ}(p)}, \circ_m^{\mathtt{pc+1}}) \in R$.

The case where there are two assignments also has $(\circ_m^p, \circ_m^{\mathtt{pc}}) \in R$. There exists $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ}(p)}$, and there is also $\circ_m^{\mathtt{pc}} \implies \circ_m^{\mathtt{pc+2}}$, which transverses $\circ_m^{\mathtt{pc+1}}$. Then also $(\circ_m^{\mathtt{succ}(p)}, \circ_m^{\mathtt{pc+2}}) \in R$.

The case for `nop` is analogous to the case of `store` which produces a single instruction.

Case relevant instruction $i \in$ JmpInst:

The only relevant instruction in this subset is `goto` q. The direct extraction produces a single transition from one normal node to the normal node tagged with the q position:

$$\mathtt{m}\mathcal{G}((p,\mathtt{goto}\ \ q),H) = \{\ \circ_m^p \xrightarrow{\varepsilon} \circ_m^q\ \}$$

The transformation $\mathtt{BC2BIR}_{instr}$ also returns a single instruction, which applied to $\mathtt{b}\mathcal{G}$ function produces a single transition:

$$\texttt{BC2BIR}_{instr}(p, \texttt{goto } q) = \texttt{[goto pc']}$$

$$\texttt{b}\mathcal{G}(\texttt{[goto pc']}_{\texttt{pc}}, \bar{H}) = \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc'}}\}$$

The case for the nodes tagged with irrelevant instructions was explained in the CmpInst case. Thus, for all nodes $\circ_m^i$ in the path graph extracted from the irrelevant instructions, we have that $(\circ_m^i, \circ_m^{\texttt{pc}}) \in R$.

Next, we analyze the relevant instruction. We have that $(\circ_m^p, \circ_m^{\texttt{pc}}) \in R$. There exists the transition $\circ_m^p \xrightarrow{\varepsilon} \circ_m^q$, and there is also $\circ_m^{\texttt{pc}} \implies \circ_m^{\texttt{pc'}}$, Thus $(\circ_m^q, \circ_m^{\texttt{pc'}}) \in R$.

Case relevant instruction $i \in$ CndInst:

The only relevant instruction in this subset is `if q`. The direct extraction produces two transitions from the normal node tagged with position p:

$$\texttt{m}\mathcal{G}((p, \texttt{if } q), H) = \{\ \circ_m^p \xrightarrow{\varepsilon} \circ_m^{\texttt{succ}(p)},\ \circ_m^p \xrightarrow{\varepsilon} \circ_m^q\ \}$$

The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction, which applied to $\texttt{b}\mathcal{G}$ function produces two transitions:

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

$$\texttt{b}\mathcal{G}(\texttt{[if } \textit{expr} \texttt{ pc']}_{\texttt{pc}}, \bar{H}) = \{\ \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc'}+1},\ \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc'}}\ \}$$

As mentioned before, for all nodes $\circ_m^i$ in the path graph extracted from the irrelevant instructions, we have that $(\circ_m^i, \circ_m^{\texttt{pc}}) \in R$. Also, the last transition in the path is $\circ_m^i \xrightarrow{\varepsilon} \circ_m^p$.

From the relevant instruction we have that $(\circ_m^p, \circ_m^{\texttt{pc}}) \in R$. There is the transition $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{\texttt{succ}(p)}$, and there is also $\circ_m^{\texttt{pc}} \implies \circ_m^{\texttt{pc}+1}$. Thus $(\circ_m^{\texttt{succ}(p)}, \circ_m^{\texttt{pc}+1}) \in R$. There is a second transition from the same source node: $\circ_m^p \xrightarrow{\varepsilon} \circ_m^q$. There is also $\circ_m^{\texttt{pc}} \implies \circ_m^{\texttt{pc'}}$, and $(\circ_m^q, \circ_m^{\texttt{pc'}}) \in R$.

Case relevant instruction $i = \texttt{throw } X$:

As defined previously, X denotes the set containing the static type of the exception being thrown, and its subtypes. Such set is the same for the direct or indirect extractions. So we present the proof for an arbitrary exception $x \in X$, and generalize the result to all the other elements.

The rule in the direct extraction for the `throw` instruction produces two edges. However, the sink node varies, in case the exception x is caught within the same method it was raised, or not.

$$\texttt{m}\mathcal{G}((p, \texttt{throw}), H) = \begin{cases} \{\ \circ_m^p \xrightarrow{handle} \bullet_m^{p,x},\quad \bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^q\ \} & \text{if has handler} \\ \{\ \circ_m^p \xrightarrow{handle} \bullet_m^{p,x}, \bullet_m^{p,x} \xrightarrow{handle} \bullet_m^{p,x,r}\} & \text{otherwise} \end{cases}$$

The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction:

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

The $\texttt{b}\mathcal{G}$ function produces two transitions, similarly to $\texttt{m}\mathcal{G}$. The second transition depends in the presence of a handler for the exception x in position pc:

$$\texttt{b}\mathcal{G}(\texttt{[throw x]}_{\texttt{pc}}, \bar{H}) = \begin{cases} \{\ \circ_m^{\texttt{pc}} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc},x},\quad \bullet_m^{\texttt{pc},x} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc'}}\ \} & \text{if has handler} \\ \{\ \circ_m^{\texttt{pc}} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc},x}, \bullet_m^{\texttt{pc},x} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc},x,r}\ \} & \text{otherwise} \end{cases}$$

Again, for all nodes $\circ_m^i$ in the path graph extracted from the irrelevant instructions, we have that $(\circ_m^i, \circ_m^{\texttt{pc}}) \in R$, and the last transition in the path is $\circ_m^i \overset{\varepsilon}{\rightarrow} \circ_m^p$.

Next, we have that $(\circ_m^p, \circ_m^{\texttt{pc}}) \in R$. There is the transition $\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x}$, and there is also $\circ_m^{\texttt{pc}} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc},x}$. Thus $(\bullet_m^{p,x}, \bullet_m^{\texttt{pc},x}) \in R$.

Now, there are two possibilities for transitions, depending if there is an exception handler for x in p. If there is a handler, then exists $\bullet_m^{p,x} \overset{\varepsilon}{\rightarrow} \circ_m^q$. Moreover, there is also $\bullet_m^{\texttt{pc},x} \implies \circ_m^{\texttt{pc'}}$, and $(\circ_m^q, \circ_m^{\texttt{pc'}}) \in R$.

If there is no exception handler for x, then exists $\bullet_m^{p,x} \overset{handle}{\rightarrow} \bullet_m^{p,x,r}$. There is also $\bullet_m^{\texttt{pc},x} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc},x,r}$, and $(\bullet_m^{p,x,r}, \bullet_m^{\texttt{pc},x,r}) \in R$.

Case relevant instruction $i \in$ XmpInst:

The instructions in this set follow to the next control point in case they terminate the execution normally, or can raise an exception if some condition was violated.

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

Next we present this case for the `div` instruction, which can only raise the *x=ArithmeticException* (given the set $\mathcal{RE}$). The case for other instructions in XmpInst is analogous. The direct extraction produces the following set of edges:

$$\texttt{m}\mathcal{G}((p, \texttt{div}), H) = \begin{cases} \{\ \circ_m^p \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{succ}(p)}, \circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x},\quad \bullet_m^{p,x} \overset{\varepsilon}{\rightarrow} \circ_m^q\ \} & \text{if has handler} \\ \{\ \circ_m^p \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{succ}(p)}, \circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x}, \bullet_m^{p,x} \overset{handle}{\rightarrow} \bullet_m^{p,x,r}\ \} & \text{otherwise} \end{cases}$$

The $\texttt{BC2BIR}_{instr}$ transformation returns a single instruction, which is an assertion:

$$\texttt{BC2BIR}_{instr}(p, \texttt{div}) = \texttt{[notzero]}$$

The $\texttt{b}\mathcal{G}$ function produces three transitions: one to a normal node, denoting absence of exceptions, one to exceptional node, denoting the transfer of control to the JVM. The third transition varies if an exception handler is found. Thus we may have two sets of transitions:

$$\mathtt{b}\mathcal{G}(\mathtt{[notzero]}_{\mathtt{pc}}, \bar{H}) = \{\ \circ_m^{\mathtt{pc}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc}+1},\ \circ_m^{\mathtt{pc}} \overset{handle}{\rightarrow} \bullet_m^{\mathtt{pc},x},\ \bullet_m^{\mathtt{pc},x} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc'}}\ \}$$

or the following, in case there is no handler for x:

$$\mathtt{b}\mathcal{G}(\mathtt{[notzero]}_{\mathtt{pc}}, \bar{H}) = \{\ \circ_m^{\mathtt{pc}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc}+1},\ \circ_m^{\mathtt{pc}} \overset{handle}{\rightarrow} \bullet_m^{\mathtt{pc},x},\ \bullet_m^{\mathtt{pc},x} \overset{handle}{\rightarrow} \bullet_m^{\mathtt{pc},x,r}\ \}$$

Again, for all nodes $\circ_m^i$ in the path graph extracted from the irrelevant instructions, we have that $(\circ_m^i, \circ_m^{\mathtt{pc}}) \in R$, and the last transition in the path is $\circ_m^i \xrightarrow{\varepsilon} \circ_m^p$.

From the relevant instruction we have that $(\circ_m^p, \circ_m^{\mathtt{pc}}) \in R$. There is the transition $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{\mathtt{succ}(p)}$, and there is also $\circ_m^{\mathtt{pc}} \implies \circ_m^{\mathtt{pc}+1}$. Thus $(\circ_m^{\mathtt{succ}(p)}, \circ_m^{\mathtt{pc}+1}) \in R$. There is a second transition from the same source node: $\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x}$. There is also $\circ_m^{\mathtt{pc}} \overset{handle}{\Longrightarrow} \bullet_m^{\mathtt{pc},x}$, and $(\bullet_m^{p,x}, \bullet_m^{\mathtt{pc},x}) \in R$.

There are two possibilities for the third transition, depending if there is an exception handler for x in p. If there is a handler, then exists $\bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^q$. Moreover, there is also $\bullet_m^{\mathtt{pc},x} \implies \circ_m^{\mathtt{pc'}}$, and $(\circ_m^q, \circ_m^{\mathtt{pc'}}) \in R$.

If there is no exception handler for x, then exists $\bullet_m^{p,x} \overset{handle}{\rightarrow} \bullet_m^{p,x,r}$. There is also $\bullet_m^{\mathtt{pc},x} \overset{handle}{\Longrightarrow} \bullet_m^{\mathtt{pc},x,r}$, and $(\bullet_m^{p,x,r}, \bullet_m^{\mathtt{pc},x,r}) \in R$.

Case relevant instruction $i \in$ InvInst:

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

Case relevant instruction $i =$ `invokespecial`:

We start with the case of `invokespecial`, which calls methods that belong to current class (including object constructors), or to the super class. The direct algorithm extracts a variable number of edges. It produces a minimum of three: one edge for the normal execution of the method, and two edges for the exceptional flow of $\varrho =$ *NullPointerException* (control transfer to JVM and exception handling).

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

The direct algorithm may extract the following set of edges:

$$\mathtt{m}\mathcal{G}((p, \mathtt{invokespecial}), H) =$$
$$\begin{cases} \{\circ_m^p \xrightarrow{\mathtt{C()}} \circ_m^{\mathtt{succ}(p)}, \circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,\varrho}, \bullet_m^{p,\varrho} \xrightarrow{\varepsilon} \circ_m^q\} \cup \mathcal{N}_p^i & \text{if } \varrho \text{ has handler} \\ \{\circ_m^p \xrightarrow{\mathtt{C()}} \circ_m^{\mathtt{succ}(p)}, \circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,\varrho}, \bullet_m^{p,\varrho} \overset{handle}{\rightarrow} \bullet_m^{p,\varrho,r}\} \cup \mathcal{N}_p^i & \text{otherwise} \end{cases}$$

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

$$\mathcal{N}_p^i = \begin{cases} \{\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x}, \bullet_m^{p,x} \overset{\varepsilon}{\rightarrow} \circ_m^q\} & \text{If has handler} \\ \{\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x}, \bullet_m^{p,x} \overset{handle}{\rightarrow} \bullet_m^{p,x,r}\} & \text{otherwise} \end{cases}$$

The $\texttt{BC2BIR}_{instr}$ transformation can return two different sets of instructions for the `invokespecial`. First, we present the case for the invocation of object constructor. It returns a sequence of assignments to temporary variables ($\texttt{[t}^0_{\texttt{pc}}\texttt{:=x]}$), denoted by $HSave$; plus the call to `[new C]`:

$$\texttt{BC2BIR}_{instr}(p, \texttt{invokespecial}) = \texttt{[}HSave\texttt{(pc,}as\texttt{);t}^0_{\texttt{pc}}\texttt{:=new C(...)]}$$

Assignments to variables produce a single transition to the next control point. Thus, the extraction of $HSave$ function produces a path graph:

$$\texttt{b}\mathcal{G}(HSave\texttt{(pc,}as\texttt{)}_{\texttt{pc}}, \bar{H}) = \{\circ_m^{\texttt{pc}} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc+1}} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc+2}}, \ldots, \circ_m^{\texttt{pc'}-1} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc'}}\}$$

The rule for the `[new C]` produces one normal edge for the case of successful execution, one pair of edges relative to the exceptional flow in case exception ϱ(=*NullPointerException*), and a pair of edges for the propagation of exception x (denoted by $\bar{\mathcal{N}}^n_{\texttt{pc}}$):

$$\texttt{b}\mathcal{G}(\texttt{[t}^0_{\texttt{pc'}}\texttt{:=new C(...)]}_{\texttt{pc'}}, \bar{H}) =$$
$$\begin{cases} \{\circ_m^{\texttt{pc'}} \overset{\texttt{C()}}{\rightarrow} \circ_m^{\texttt{pc'+1}},\ \circ_m^{\texttt{pc'}} \overset{\varepsilon}{\rightarrow} \bullet_m^{\texttt{pc'},\varrho},\ \bullet_m^{\texttt{pc'},\varrho} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc''}}\} \cup \bar{\mathcal{N}}^{\texttt{C}}_{\texttt{pc}} & \text{If } \varrho \text{ has handler} \\ \{\circ_m^{\texttt{pc'}} \overset{\texttt{C()}}{\rightarrow} \circ_m^{\texttt{pc'+1}},\ \circ_m^{\texttt{pc'}} \overset{\varepsilon}{\rightarrow} \bullet_m^{\texttt{pc'}\varrho},\ \bullet_m^{\texttt{pc'},\varrho} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc'},\varrho,r}\} \cup \bar{\mathcal{N}}^{\texttt{C}}_{\texttt{pc}} & \text{otherwise} \end{cases}$$

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

$$\bar{\mathcal{N}}^n_{\texttt{pc}} = \begin{cases} \{\circ_m^{\texttt{pc'}} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc'},x},\ \bullet_m^{\texttt{pc'},x} \overset{handle}{\rightarrow} \circ_m^{\texttt{pc''}}\} & \text{If has handler} \\ \{\circ_m^{\texttt{pc'}} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc'},x},\ \bullet_m^{\texttt{pc'},x} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc'},x,r}\} & \text{otherwise} \end{cases}$$

Again, for all nodes $\circ_m^i$ in the path graph extracted from the irrelevant instructions, we have that $(\circ_m^i, \circ_m^{\texttt{pc}}) \in R$, and the last transition in the path is $\circ_m^i \overset{\varepsilon}{\rightarrow} \circ_m^p$.

Next we analyze the relevant instruction. $(\circ_m^p, \circ_m^{\texttt{pc}}) \in R$. There is the transition $\circ_m^p \overset{C()}{\rightarrow} \circ_m^{\texttt{succ}(p)}$, and there is also $\circ_m^{\texttt{pc}} \overset{C()}{\Longrightarrow} \circ_m^{\texttt{pc''}}$, which transverses all the nodes produces from $HSave$. Thus $(\circ_m^{\texttt{succ}(p)}, \circ_m^{\texttt{pc''}}) \in R$.

There is a second transition from the same source node: $\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,\varrho}$. There is also $\circ_m^{\texttt{pc}} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc'},\varrho}$, which also transverses nodes produces from $HSave$, and

$(\bullet_m^{p,\varrho}, \bullet_m^{\texttt{pc'},\varrho}) \in R$. The next edge depends on the presence of a handler. If there is none, then exists a transition $\bullet_m^{p,\varrho} \overset{handle}{\rightarrow} \bullet_m^{p,\varrho,r}$. There is also $\bullet_m^{\texttt{pc'},\varrho} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc'},\varrho,\texttt{r}}$ and $(\bullet_m^{p,\varrho,r}, \bullet_m^{\texttt{pc'},\varrho,\texttt{r}}) \in R$. If there is a handler, then $(\circ_m^q, \circ_m^{\texttt{pc'''}}) \in R$, and explanation is analogous. There is also the pair of transitions added by the propagation of exception x, which is analogous: the first transition being into an exceptional node, and the second varies according to having a suitable handler for x.

The second case for `invokespecial` is the one where the called method is not a constructor, but some method within the same class, or from the super class.

The $\texttt{BC2BIR}_{instr}$ transformation returns the same instructions as before, but preceded by `[notnull]` instruction.

$$\texttt{BC2BIR}_{instr}(p, \texttt{invokespecial}) = \texttt{[notnull];[}\mathit{HSave}\texttt{(pc,}as\texttt{);t}^0_{\texttt{pc}}\texttt{:=new C(...)]}$$

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

$$\texttt{b}\mathcal{G}(\texttt{[notnull]}_{\texttt{pc}}, \bar{H}) = \begin{cases} \{\ \circ_m^{\texttt{pc}} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc+1}},\ \circ_m^{\texttt{pc}} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc},\varrho},\ \bullet_m^{\texttt{pc},\varrho} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc'}}\ \} & \text{If } \varrho \text{ has handler} \\ \{\ \circ_m^{\texttt{pc}} \overset{\varepsilon}{\rightarrow} \circ_m^{\texttt{pc+1}},\ \circ_m^{\texttt{pc}} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc},\varrho},\ \bullet_m^{\texttt{pc},\varrho} \overset{handle}{\rightarrow} \bullet_m^{\texttt{pc},\varrho,r}\ \} & \text{otherwise} \end{cases}$$

The case for all nodes $\circ_m^i$ in the path graph extracted from the irrelevant instructions is the same, and we have that $(\circ_m^i, \circ_m^{\texttt{pc}}) \in R$, and the last transition in the path is $\circ_m^i \overset{\varepsilon}{\rightarrow} \circ_m^p$.

Next we analyze the relevant instruction. $(\circ_m^p, \circ_m^{\texttt{pc}}) \in R$. There is the transition $\circ_m^p \overset{n()}{\rightarrow} \circ_m^{\texttt{succ}(p)}$, and there is also $\circ_m^{\texttt{pc}} \overset{n()}{\Longrightarrow} \circ_m^{\texttt{pc''}}$, which transverses the node tagged with `notnull` position, and all the nodes produces from HSave. Thus $(\circ_m^{\texttt{succ}(p)}, \circ_m^{\texttt{pc''}}) \in R$.

There is a second transition from the same source node: $\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,\varrho}$. There is also $\circ_m^{\texttt{pc}} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc},\varrho}$, containing the edge produced by `[notnull]` and $(\bullet_m^{p,\varrho}, \bullet_m^{\texttt{pc},\varrho}) \in R$. Again, the next edge varies on the presence of a handler or not. If there is none, then exists a transition $\bullet_m^{p,\varrho} \overset{handle}{\rightarrow} \bullet_m^{p,\varrho,r}$. There is also $\bullet_m^{\texttt{pc'},\varrho} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc},\varrho,\texttt{r}}$ and $(\bullet_m^{p,\varrho,r}, \bullet_m^{\texttt{pc},\varrho,\texttt{r}}) \in R$. If there is a handler, then $\bullet_m^{p,\varrho} \overset{\varepsilon}{\rightarrow} \circ_m^q$. Moreover, there is $\bullet_m^{\texttt{pc'},\varrho} \overset{handle}{\Longrightarrow} \circ_m^{\texttt{pc'''}}$ Therefore, $(\circ_m^q, \circ_m^{\texttt{pc'''}}) \in R$.

Again, the explanation transitions of propagation of exception x is analogous to the case for ϱ. There is a third transition from the same source node: $\circ_m^p \overset{handle}{\rightarrow} \bullet_m^{p,x}$. There is also $\circ_m^{\texttt{pc}} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc'},\texttt{x}}$, which transverse the node produced by `[notnull]`, and the nodes produced by HSave. and $(\bullet_m^{p,x}, \bullet_m^{\texttt{pc'},\texttt{x}}) \in R$. If there is no handler for x, then exists a transition $\bullet_m^{p,x} \overset{handle}{\rightarrow} \bullet_m^{p,x,r}$. There is also $\bullet_m^{\texttt{pc'},x} \overset{handle}{\Longrightarrow} \bullet_m^{\texttt{pc'},\texttt{x},\texttt{r}}$ and $(\bullet_m^{p,x,r}, \bullet_m^{\texttt{pc},x,r}) \in R$. If there is a handler, then $\bullet_m^{p,x} \overset{\varepsilon}{\rightarrow} \circ_m^q$. Moreover, there is $\bullet_m^{\texttt{pc'},x} \overset{handle}{\Longrightarrow} \circ_m^{\texttt{pc'''}}$ Therefore, $(\circ_m^q, \circ_m^{\texttt{pc'''}}) \in R$.

Case relevant instruction $i = \texttt{invokevirtual}$:

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

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

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

$$\texttt{m}\mathcal{G}((p, \texttt{invokevirtual}), H) = \begin{cases} \{\circ_m^p \xrightarrow{n()} \circ_m^{\texttt{succ(}p\texttt{)}}, \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\varrho}, \bullet_m^{p,\varrho} \xrightarrow{\varepsilon} \circ_m^q\} \cup \mathcal{N}_p^i & \text{If } \varrho \text{ has handler} \\ \{\circ_m^p \xrightarrow{n()} \circ_m^{\texttt{succ(}p\texttt{)}}, \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\varrho}, \bullet_m^{p,\varrho} \xrightarrow{handle} \bullet_m^{p,\varrho,r}\} \cup \mathcal{N}_p^i & \text{otherwise} \end{cases}$$

The function $\mathcal{N}_p^i$ produces the following set of nodes for some exception x propagated from a call to $n()$:

$$\mathcal{N}_p^i = \begin{cases} \{\circ_m^p \xrightarrow{handle} \bullet_m^{p,x}, \bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^q\} \cup \mathcal{N}_p^i & \text{If has handler} \\ \{\circ_m^p \xrightarrow{handle} \bullet_m^{p,x}, \bullet_m^{p,x} \xrightarrow{handler} \bullet_m^{p,x,r}\} \cup \mathcal{N}_p^i & \text{otherwise} \end{cases}$$

The $\texttt{BC2BIR}_{instr}$ outputs a set of instructions, being the minimum two: the assertion `[notnull]` and the method invocation:

$$\texttt{BC2BIR}_{instr}(p, \texttt{invokevirtual}) = \texttt{[notnull;}\mathit{HSave}\texttt{(pc,}as\texttt{);t}^0_{\texttt{pc}}\texttt{:=e.m(...)]}$$

Applying the extraction function to `[notnull]` we get the following edges:

$$\texttt{b}\mathcal{G}(\texttt{[notnull]}_{\texttt{pc}}, \bar{H}) = \begin{cases} \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{handle} \bullet_m^{\texttt{pc},\varrho}, \bullet_m^{\texttt{pc},\varrho} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc''}}\} & \text{If } \varrho \text{ has handler} \\ \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{handle} \bullet_m^{\texttt{pc},\varrho}, \bullet_m^{\texttt{pc},\varrho} \xrightarrow{handle} \bullet_m^{\texttt{pc},\varrho,r}\} & \text{otherwise} \end{cases}$$

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

$$\texttt{b}\mathcal{G}(\mathit{HSave}(\texttt{pc}, as)_{\texttt{pc}}, \bar{H}) = \{\circ_m^{\texttt{pc+1}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+2}}, \ldots, \circ_m^{\texttt{pc'}-1} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc'}}\}$$

The rule for the $\texttt{[t}^0_{\texttt{pc}}\texttt{,:=e.n(...)]}$ produces one normal edge for the case of successful execution, and a pair of edges for the propagation of exception x (denoted by $\bar{\mathcal{N}}_{\texttt{pc}}^n$):

$$\mathtt{b}\mathcal{G}([\mathtt{t}^0_{\mathtt{pc'}},\mathtt{:=e.n(...)}]_{\mathtt{pc}},\bar{H}) = \{\ \circ^{\mathtt{pc'}}_m \overset{n()}{\rightarrow} \circ^{\mathtt{pc'}+1}_m\ \} \cup \bar{\mathcal{N}}^n_{\mathtt{pc}}$$

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

$$\bar{\mathcal{N}}^n_{\mathtt{pc}} = \begin{cases} \{\circ^{\mathtt{pc'}}_m \overset{handle}{\rightarrow} \bullet^{\mathtt{pc',x}}_m,\ \bullet^{\mathtt{pc',x}}_m \overset{handle}{\rightarrow} \circ^{\mathtt{pc'',x,r}}_m\} & \text{If has handler} \\ \{\circ^{\mathtt{pc'}}_m \overset{handle}{\rightarrow} \bullet^{\mathtt{pc',x}}_m,\ \bullet^{\mathtt{pc',x}}_m \overset{handle}{\rightarrow} \bullet^{\mathtt{pc',x,r}}_m\} & \text{otherwise} \end{cases}$$

The case for all nodes $\circ^i_m$ in the path graph extracted from the irrelevant instructions is the same, and we have that $(\circ^i_m, \circ^{\mathtt{pc}}_m) \in R$, and the last transition in the path is $\circ^i_m \overset{\varepsilon}{\rightarrow} \circ^p_m$.

Next we analyze the relevant instruction. $(\circ^p_m, \circ^{\mathtt{pc}}_m) \in R$. There is the transition $\circ^p_m \overset{n()}{\rightarrow} \circ^{\mathtt{succ}(p)}_m$, and there is also $\circ^{\mathtt{pc}}_m \overset{n()}{\Longrightarrow} \circ^{\mathtt{pc''}}_m$, which transverses the node tagged with `notnull` position, and all the nodes produces from $HSave$. Thus $(\circ^{\mathtt{succ}(p)}_m, \circ^{\mathtt{pc''}}_m) \in R$.

There is a second transition from the same source node: $\circ^p_m \overset{handle}{\rightarrow} \bullet^{p,\varrho}_m$. There is also $\circ^{\mathtt{pc}}_m \overset{handle}{\Longrightarrow} \bullet^{\mathtt{pc},\varrho}_m$, containing the edge produced by `[notnull]` and $(\bullet^{p,\varrho}_m, \bullet^{\mathtt{pc},\varrho}_m) \in R$. Again, the next edge varies on the presence of a handler or not. If there is none, then exists a transition $\bullet^{p,\varrho}_m \overset{handle}{\rightarrow} \bullet^{p,\varrho,r}_m$. There is also $\bullet^{\mathtt{pc'},\varrho}_m \overset{handle}{\Longrightarrow} \bullet^{\mathtt{pc},\varrho,\mathtt{r}}_m$ and $(\bullet^{p,\varrho,r}_m, \bullet^{\mathtt{pc},\varrho,\mathtt{r}}_m) \in R$. If there is a handler, then $\bullet^{p,\varrho}_m \overset{\varepsilon}{\rightarrow} \circ^q_m$. Moreover, there is $\bullet^{\mathtt{pc'},\varrho}_m \overset{handle}{\Longrightarrow} \circ^{\mathtt{pc'''}}_m$ Therefore, $(\circ^q_m, \circ^{\mathtt{pc'''}}_m) \in R$.

Again, the explanation transitions of propagation of exception x is analogous to the case for ϱ. There is a third transition from the same source node: $\circ^p_m \overset{handle}{\rightarrow} \bullet^{p,x}_m$. There is also $\circ^{\mathtt{pc}}_m \overset{handle}{\Longrightarrow} \bullet^{\mathtt{pc',x}}_m$, which transverse the node produced by `[notnull]`, and the nodes produced by $HSave$. and $(\bullet^{p,x}_m, \bullet^{\mathtt{pc',x}}_m) \in R$. If there is no handler for x, then exists a transition $\bullet^{p,x}_m \overset{handle}{\rightarrow} \bullet^{p,x,r}_m$. There is also $\bullet^{\mathtt{pc'},x}_m \overset{handle}{\Longrightarrow} \bullet^{\mathtt{pc',x,r}}_m$ and $(\bullet^{p,x,r}_m, \bullet^{\mathtt{pc},x,r}_m) \in R$. If there is a handler, then $\bullet^{p,x}_m \overset{\varepsilon}{\rightarrow} \circ^q_m$. Moreover, there is $\bullet^{\mathtt{pc'},x}_m \overset{handle}{\Longrightarrow} \circ^{\mathtt{pc'''}}_m$ Therefore, $(\circ^q_m, \circ^{\mathtt{pc'''}}_m) \in R$.

□

6 Related Work

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

Sinha *et al.* [16, 17] propose a control-flow graph extraction algorithm for both Java source and bytecode, which takes into account *explicit* exceptions only. The algorithm performs first an intra-procedural analysis, computing the

exceptional return nodes caused by uncaught exceptions. Next, it executes an inter-procedural analysis to compute exception propagation paths. This division is similar to how our algorithm analyses exceptional flows, using a slightly different inter-procedural analysis. However, the authors do not discuss how the static type of explicit exceptions is determined by the bytecode analysis, whereas we get this information from the BIR transformation. Moreover, the use of BIR allows us to also support (a subset of the) *implicit* exceptions.

Jiang *et al.* [13] extend the work of Sinha *et al.* to C++ source code. C++ has the same scheme of `try-catch` and exception propagation as Java, but without the `finally` blocks, or implicit exceptions. This work does not consider the exceptions types. Thus, it heavily over-approximates the possible flows by connecting the control points with explicit `throw` within a `try` block to all its `catch` blocks, and considering that any called method containing a `throw` may terminate exceptionally. Our work consider the exceptions types. Thus, produce more refined CFGs, and also tells which exceptions can be raised, or propagated from method invocations.

Choi *et al.* [6] use an intermediate representation from the Jalapeño compiler [5] to extract CFGs with exceptional flows. The authors introduce a stack-less representation, using assertions to mark the possibility of an instruction raising an exception. This approach was followed by Demange *et al.* when defining BIR, and proving the correctness of the transformation from bytecode. As a result, our extraction algorithm, via BIR, is very similar to that of Choi. We differ by defining formal extraction rules, and proving its correctness w.r.t. behavior.

Finally, Jo and Chang [14] construct CFGs from Java source code by computing normal and exceptional flows separately. An iterative fixed-point computation is then used to merge the exceptional and the normal control-flow graphs. Our exception propagation computation follows their approach; however, the authors do not discuss how the exception type is determined. Also, only explicit exceptions are supported; in contrast, we determine the exception type and support implicit exceptions by using the BIR transformation.

7 Conclusion

This paper presents an efficient and precise control-flow graph extraction algorithm that also considers exceptions. It presents a formal argument why the algorithm is correct, i.e., it extracts a graph whose behavior over-approximates the behavior of the original program. To the best of our knowledge, this is the first CFG extraction algorithm that has been proven correct. The proof is presented in pencil-and-paper style, but paves the ground for a mechanized proof using a standard theorem prover.

The algorithm is precise because it uses BIR, an intermediate stack-less representation. The BIR transformation provides precise information about exceptional control-flow, and at the same time it makes the generated control-flow graphs relatively small.

To prove correctness of the algorithm, a second idealized extraction algorithm that works directly on the bytecode is presented. It is easy to prove correctness of this direct algorithm. To prove correctness of the indirect algorithm we show that the resulting CFG structurally simulates the CFG generated by the direct algorithm. Since structural simulation implies behavioral simulation, this gives us the desired result.

The extraction was implemented as the CFGEX tool. The experimental results show that the algorithm is efficient, and that it produces compact CFGs.

Future Work Currently we study how to adapt the algorithm to a modular setting. Our intention is to use the extracted CFGs as input for CVPP [12], a tool set for compositional verification of control-flow safety properties. In this setting, one typically wishes to produce CFGs from incomplete programs.

In addition, we also plan to study how the algorithm can be adapted to preserve some data of the original program, and how to use it for programs with multiple threads of execution.

Acknowledgments We thank the Celtique team at INRIA Rennes for their help on the BIR language.

References

1. 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
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA. pp. 324–341 (1996)
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. Burke, M.G., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño dynamic optimizing compiler for Java. In: Proceedings of the ACM 1999 conference on Java Grande. pp. 129–141. JAVA '99, ACM, New York, NY, USA (1999)
6. Choi, J.D., Grove, D., Hind, M., Sarkar, V.: Efficient and precise modeling of exceptions for the analysis of Java programs. SIGSOFT Softw. Eng. Notes 24, 21–31 (September 1999)
7. 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
8. Freund, S.N., Mitchell, J.C.: A type system for the Java bytecode language and verifier. J. Autom. Reason. 30, 271–321 (August 2003)
9. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)

10. 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)
11. 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)
12. 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)
13. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. SIGPLAN Not. 42, 3–8 (April 2007)
14. 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)
15. Milner, R.: Communicating and mobile systems: the π-calculus, chap. 6, pp. 52–53. Cambridge University Press, New York, NY, USA (1999)
16. 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. 265–276. ICSM '99, IEEE Computer Society (1999)
17. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. 26, 849–871 (September 2000)