Sound Control-Flow Graph Extraction for 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
[2] KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. We present an algorithm to extract control-flow graphs from Java bytecode, considering exceptional flows. We then establish its correctness: the behavior of the extracted graphs is shown to be a sound over-approximation of the behavior of the original programs. Thus, any temporal safety property that holds for the extracted control-flow graph also holds for the original program. This makes the extracted graphs suitable for performing various static analyses, in particular model checking. The extraction proceeds in two phases. First, we translate Java bytecode into BIR, a stack-less intermediate representation. The BIR transformation is developed as a module of Sawja, a novel static analysis framework for Java bytecode. Besides Sawja's efficiency, the resulting intermediate representation is more compact than the original bytecode and provides an explicit representation of exceptions. These features make BIR a natural starting point for sound control-flow graph extraction. Next, we formally define the transformation from BIR to control-flow graphs, which (among other features) considers the propagation of uncaught exceptions within method calls. We prove the correctness of the two-phase extraction by suitably combining the properties of the two transformations with those of an idealized control-flow graph extraction algorithm, whose correctness has been proved directly. The control-flow graph extraction algorithm is implemented in the ConFlEx tool. A number of test-cases show the efficiency and the utility of the implementation.

1 Introduction

Over the last decade, there has been a steadily increasing demand for software quality and reliability. Different formal techniques have been deployed to reach this goal, such as various static analyses, model checking and (automated) theorem proving. A major obstacle for the application of formal techniques is that the state space of software is typically infinite. Appropriate abstractions are thus necessary in order to make the formal analyses tractable. Further, 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 approach is to generate an abstract model from the code, only preserving the information that is relevant for the class of properties of interest. In particular, *control-flow graphs* (CFGs) are a widely used abstraction, where only the control-flow information is kept, and all program data is abstracted away (see *e.g.* [6,19,16]). In a CFG, nodes represent the control points of the program, while edges represent the instructions that move control between control points.

Numerous techniques have been proposed to extract automatically control-flow graphs from program code (see *e.g.* [15,8,16]). Typically, however, these are not accompanied by a formal soundness argument. The present paper attempts to fill this gap: we define a control-flow graph extraction algorithm for sequential Java bytecode (JBC), and show that the extraction algorithm is sound w.r.t. the behavior (*i.e.*, executions) of the program. The extraction algorithm considers all the typical intricacies of Java, such as virtual method call resolution, the differences between dynamic and static object types, and exception handling. In particular, it includes explicitly thrown instructions, and a significant subset of run-time exceptions. The sound analysis of exceptional flows is particularly challenging for two reasons. First, the stack-based nature of the Java Virtual Machine (JVM) makes it hard to statically 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*, and to keep track of where such exceptions can be raised requires much care.

We present a two-phase extraction algorithm using the Bytecode Intermediate Representation (BIR) language [9], developed by Demange *et al.* The use of BIR has several advantages. First of all, BIR provides a stack-less representation of JBC. Thus, all instructions (including the explicit `athrow`) are directly connected with their operands. This allows to determine the static type of explicitly thrown exceptions. In addition, the representation of a program in BIR is smaller, since operations are not stack-based, but represented as expression trees. Second, BIR supports the analysis of implicitly thrown exceptions by generating assertions that indicate when the next instruction might raise a run-time exception, following the approach proposed for the Jalapeño compiler [7]. Finally, Demange *et al.* present formal translation rules from JBC, and define an operational semantics for BIR. They show that the resulting program is semantics-preserving with respect to observable events, such as raising exceptions, and sequences of method invocations. This result increases the reliability of the correctness of the BIR transformation, and in consequence, also of our CFG extraction algorithm.

Our two-phase extraction algorithm first uses the transformation from JBC to BIR from Sawja [11], a library for static analysis of Java bytecode, and then it extracts CFGs from BIR. It is implemented as the tool ConFlEx. Sawja provides only intra-procedural support for exceptions. Thus, to obtain a sound extraction tool, on top of this we implemented a fixed-point computation of exceptional flow caused by uncaught exceptions.

Proving correctness of the two-phase extraction algorithm directly (*e.g.*, by means of behavioral simulation) is cumbersome. Instead, we use the correctness of an idealized direct extraction algorithm by Amighi [2,3] to simplify the overall correctness argument. We connect the CFGs that are extracted by the idealized algorithm and by the two-phase algorithm via a (structural) simulation relation, and use a previous result (see [10, Th. 36]) to infer behavioral simulation. From this, one 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. We outline the correctness proof in Section 4; the details can be found in an accompanying technical report [3].

Organization. The remainder of this paper is organized as follows. Section 2 provides the necessary definitions for the algorithm and its correctness proof. Section 3 presents the two-phase extraction algorithm, its implementation, and experimental evaluation. In Section 4 we discuss the correctness of the algorithm. Finally, in Section 5 we discuss related work, and conclude with Section 6.

2 Preliminaries

Control-flow graphs (CFGs) provide an abstract model of programs. Method graphs are the basic building blocks of CFGs. Let Meth and Excp be two countably infinite sets of method names and exception names, respectively. Method graphs are defined as Kripke structures, as follows.

Definition 1 (Method Graph). *A* method graph for method *m over given finite sets $M \subseteq$* Meth *and $E \subseteq$* Excp *is a pair $(\mathcal{M}_m, \mathbb{E}_m)$, where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a transition-labeled* Kripke structure*, and $\mathbb{E}_m \subseteq V_M$ is a non-empty set of* entry points *of m. V_m is the set of* control points *of m, $L_m = M \cup \{\varepsilon, handle\}$ is the set of transition labels, $\rightarrow_m \subseteq V_m \times L_m \times V_m$ is the labeled transition relation between control points, $A_m = \{m, r\} \cup E$ is the set of atomic propositions, and $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ is a valuation function such that $m \in \lambda_m(v)$ for all $v \in V_m$, and for all $x, x' \in E$, if $x, x' \in \lambda_m(v)$ then $x = x'$, i.e., each control node is valuated with the method signature it belongs to, and with at most one exception.*

A node $v \in V_m$ is marked with the atomic proposition r whenever it is a return node of the method. Internal transfer edges are labeled with ε, and the control transfers caused by the handling of exceptions are labeled with *handle*. All other edges correspond to method calls, and are labeled with the called method.

The *control-flow graph* of a program is simply the disjoint union of all method graphs of methods defined in the program. Figure 1 shows an example Java program with two methods, and a corresponding CFG. Every control-flow graph is equipped with an *interface* $I = (I^+, I^-, E')$, defining the methods that are provided to and required from the environment, denoted by $I^+, I^- \subseteq M$, and the exceptions that may be raised by each method, but not caught, indicated as $E' \subseteq E$. If $I^- \subseteq I^+$ then I is *closed.*

We use a standard notion of control-flow graph *behavior* based on pushdown automata, where configurations are pairs of control nodes and stacks of method invocations. Internal transitions are labeled with τ for normal transfers, *throw x* and *catch x* for exceptional transfers, m_1 *call* m_2 and m_1 *ret* m_2 for normal

```
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);
 }
}
```

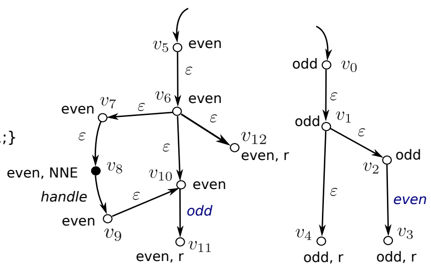

Fig. 1. An example program and its control-flow graph

inter-procedural transfers, and m_1 *xret* m_2 for returns caused by an uncaught exception. The formal definition is straightforward (see [12]), and its details are not necessary to understand the correctness proof of our extraction algorithm.

3 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 the BIR language, and its transformation function from JBC, named `BC2BIR`. Next, we present how BIR is transformed into CFGs. We conclude by describing the implementation of the algorithm as the ConFlEx tool [1], and presenting some experimental results.

3.1 The BIR Language

The BIR language is an intermediate representation of Java bytecode. The main difference with JBC is that BIR instructions are stack-less, in contrast to bytecode instructions that operate over values stored on the operand stack. We give a brief overview of BIR; for a full account we refer to [9].

Figure 2 summarizes the BIR syntax. Its instructions operate over expression trees, i.e., arithmetic expressions composed of constants, operations, variables, and fields of other expressions ($expr$.`f`). BIR does not have operations over strings and booleans; these are transformed into method 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 types of variables: *var* and *tvar*. The first are identifiers also present in the original bytecode; the latter are new variables introduced by the transformation. Both variables and object fields can be an assignment's target.

Many of the BIR instructions have an equivalent JBC counterpart, e.g., `nop`, `goto` and `if`. A `return` *expr* ends the execution of a method with return value

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

$$\begin{array}{rcl}
\mathit{Assignment} & ::= & \mathit{target} \texttt{ := } \mathit{expr} \\
\mathit{Return} & ::= & \texttt{return } \mathit{expr} \mid \texttt{return} \\
\mathit{MethodCall} & ::= & \mathit{expr}.\texttt{ns(}\mathit{expr}\texttt{,} \ldots\texttt{, } \mathit{expr}\texttt{)} \\
& \mid & \mathit{target} := \mathit{expr}.\texttt{ns(}\mathit{expr}\texttt{,} \ldots \texttt{,}\mathit{expr}\texttt{)} \\
\mathit{NewObject} & ::= & \mathit{target} := \texttt{new C(}\mathit{expr}\texttt{,} \ldots \texttt{,}\mathit{expr}\texttt{)} \\
\mathit{Assertion} & ::= & \texttt{notnull } \mathit{expr} \mid \texttt{notzero } \mathit{expr} \\
\mathit{instr} & ::= & \texttt{nop} \mid \texttt{if } \mathit{expr} \texttt{ pc} \mid \texttt{goto pc} \\
& \mid & \texttt{throw } \mathit{expr} \mid \texttt{mayinit C} \\
& \mid & \mathit{Assignment} \mid \mathit{Return} \\
& \mid & \mathit{MethodCall} \mid \mathit{NewObject} \\
& \mid & \mathit{Assertion}
\end{array}$$

Fig. 2. Expressions and Instructions of BIR

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

Fig. 3. Implicit exceptions supported by BIR, and associated assertions

expr, while `return` ends a *void* method. The `throw` instruction explicitly transfers control flow to the exception handling mechanism. Method call instructions are represented by their method signature. For non-*void* methods, the instruction assigns the result value to a variable.

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

BIR's support of implicit exceptions follows the approach proposed for the Jalapeño compiler [7]. It inserts special assertions before the instructions that can potentially raise an exception, as defined by the JVM. Figure 3 shows all implicit exceptions that are currently supported by the `BC2BIR` transformation [5], and the associated assertion. For example, the transformation inserts a `[notnull]` assertion before any instruction that might raise a *NullPointerException*, such as an access to a reference. If the assertion holds, it behaves as a `[nop]`, and control-flow passes to the next instruction. If the assertion fails, control-flow is passed to the exception handling mechanism. In the transformation from BIR to CFG, we use a function χ to obtain the exception associated with an instruction (as presented in Figure 3). Notice that our translation from BIR to CFG can easily be adapted to 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. Methods have code, stored in an

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 ns	[notnull e;*HSave*(pc,*as*);$\mathtt{t}^0_{\mathtt{pc}}$:=e.ns($\mathtt{e}'_1 \ldots \mathtt{e}'_n$)]
invokespecial ns	[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. 4. Rules for $\mathtt{BC2BIR}_{instr}$

instruction array, with indexing starting with 0 for the entry control point. However, in contrast to JBC, in BIR the indexes in the instruction array are sequential.

3.2 Transformation from Java Bytecode into BIR

Next we briefly describe 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 instructions that can be produced, we do not discuss all details of this transformation. For the complete algorithm, we refer to [9].

The symbolic execution of the individual instructions is defined by a function $\mathtt{BC2BIR}_{instr}$ that, given a program counter, a JBC instruction and an abstract stack, outputs a sequence 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.

Definition 2 (BIR Transformation Function). *Let* $AbsStack \in expr^*$. *The rules defining the* instruction-wise transformation $\mathtt{BC2BIR}_{instr} : \mathbb{N} \times instr_{JBC} \times AbsStack \rightharpoonup ((instr_{BIR})^* \times AbsStack) \cup \{Fail\}$ *from Java bytecode into BIR are given in Figure 4.*

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

JBC instructions `if`, `goto`, `return` and `vreturn` are transformed into corresponding BIR instructions. The `new` instruction 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.

Instruction `store x` produces one or two assignments, depending on the state of the abstract stack. Instruction `putfield f` outputs a set of BIR instructions: [notnull *e*] guards if *e* is a valid reference; the auxiliary function *FSave*

0:	`iload_0`		
1:	`ifne 6`	0:	`if (n != 0) goto 2`
4:	`iconst_0`		
5:	`ireturn`	1:	`return 0`
6:	`aload_0`		
7:	`iconst_1`		
8:	`isub`	2:	`mayinit Number`
9:	`invokestatic Number.even(int)`	3:	$\mathtt{t}_3^0$ `:= Number.even(n - 1)`
12:	`ireturn`	4:	`return` $\mathtt{t}_3^0$

Fig. 5. Comparison between instructions in method odd() in JBC and BIR

generates a sequence of assignments to temporary variables; followed by the assignment to the field `e.f`. Similarly, `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 case, there are assignments to temporary variables (*HSave*), followed by the instruction `[new C]`, which denotes a call to the constructor.

Figure 5 shows the JBC and BIR versions of method `odd()` from Figure 1. The different colors show the collapsing of JBC instructions by the transformation; the underlined instructions are the ones that produce BIR instructions. The BIR method has a local variable (`n`) and a newly introduced variable ($\mathtt{t}_3^0$). Notice that the argument for the method invocation and the operand to the `[if]` instruction are reconstructed expression trees. The `[mayinit]` instruction shows that class `Number` may be initialized in that program point.

3.3 Transformation from BIR into Control-Flow Graphs

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

To define $\mathtt{b}\mathcal{G}$, we introduce auxiliary definitions. First, let $\mathcal{E}_t$ be the set of all exception tables. $H \in \mathcal{E}_t$ is the exception table for the given method, containing the same entries as the JBC table, but with control points relating to BIR instructions. The function $h_H(\mathtt{pc}, x)$ searches for the first handler for the exception x (or a subtype) at position `pc`. The function $\mathcal{H}_x^{\mathtt{pc}}$ returns one edge after querying h_H: if there was an exception handler, it returns an edge to a normal control node; otherwise, it returns an edge to an exceptional return node.

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

$$\mathcal{H}_x^{\mathtt{pc}} = \begin{cases} \{\ (\bullet_m^{\mathtt{pc},x}, \mathit{handle}, \circ_m^{\mathtt{pc'}})\ \} & \text{if } h_H(\mathtt{pc}, x) = \mathtt{pc'} \neq 0 \\ \{\ (\bullet_m^{\mathtt{pc},x}, \mathit{handle}, \bullet_m^{\mathtt{pc},x,r})\ \} & \text{if } h_H(\mathtt{pc}, x) = 0 \end{cases}$$

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

$$\mathcal{N}_n^{\mathtt{pc}} = \bigcup_{\bullet_n^{\mathtt{pc'},x,r} \in \mathtt{b}\mathcal{G}(n)} \{(\circ_m^{\mathtt{pc}}, \mathit{handle}, \bullet_m^{\mathtt{pc},x})\} \cup \mathcal{H}_x^{\mathtt{pc}}$$

Fig. 6. Extraction rules for control-flow graphs from BIR

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

Definition 3 (Control Flow Graph Extraction). *The* control-flow graph extraction function $\mathtt{b}\mathcal{G} : (\mathit{Instr} \times \mathbb{N}) \times \mathcal{E}_t \to \mathcal{P}(V \times L_m \times V)$ *is defined by the rules in Figure 6. Given method* m*, with* $\mathit{ArInstr}_m$ *as its instruction array, the* control-flow graph *for* m *is defined as* $\mathtt{b}\mathcal{G}(m) = \bigcup_{i_{pc} \in \mathit{ArInstr}_m} \mathtt{b}\mathcal{G}(i_{pc}, H_m)$*, where* i_{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 for the intra-procedural analysis. 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 `[return` *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 `[throw` X`]` instruction, similarly to virtual method call resolution, depends on a static analysis to find out the possible exceptions that can be thrown. The BIR transformation only provides the static type X of the thrown exception. Let X also denote the set containing the static type, and all its subtypes. The transformation produces an exceptional edge for each element x of X, followed by the appropriate edge derived from the exception table.

The rule for assertion instructions produces a normal edge, for the case that the implicit exception is not raised, and an edge to the exceptional node tagged with the exception type (as defined in Figure 3), together with the appropriate edge derived from the exception table.

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 rule for the other method invocations adds a single normal edge for each possible receiver n returned by res^{α}.

Next, we describe the inter-procedural analysis. In all program points where there is a method invocation, the function $\mathcal{N}_n^{\mathtt{pc}}$ adds exceptional edges, relative to the exceptions that are propagated by method calls. It checks if the CFG of an invoked method n contains an exceptional return node. If it does, then function $\mathcal{H}_x^{\mathtt{pc}}$ verifies whether the exception is caught upon return. If so, it adds an edge to the handler. Otherwise it adds an edge to an exceptional return node. In the latter case, propagation of the exception continues until it is caught by a caller method, or there are no more methods to handle it. This is similar to the process described by Jo and Chang [16], who also present a fixed-point algorithm to compute the propagation edges. It checks the pre-computed call-graph which are the callers to a method propagating a given exception, and at which control-points. If there is a suitable handler for that exception, it adds the respective handling edges, and the process stop. Otherwise, the computation proceeds.

3.4 Implementation

The extraction rules from Figure 6 are implemented in our CFG extraction tool ConFlEx. It uses Sawja for the transformation from bytecode into BIR, and for virtual method call resolution. Sawja supports several resolution algorithms. Experimental evaluation showed that the algorithm's choice impacts the performance, but does not affect significantly the precision. Table 1 shows the results using Rapid Type Analysis [4], which presented the best balance between time and precision [21]. The table provides statistics for the CFG extraction of several examples with varying sizes. 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 spent to transform JBC into BIR. For the transformation from BIR to CFG, we provide statistics for the intra-procedural and the inter-procedural analysis.

Table 1 shows that in all cases the number of BIR instructions is less than 40% of the JBC instructions. This indicates that the use of BIR mitigates the blow-up of control-flow graphs, and clearly program analysis benefits from this. The computation time for intra- and inter-procedural analysis grows proportionally with the number of BIR instructions. The intra-procedural analysis is linear w.r.t. to the number of instructions, and the experimental results of the inter-procedural analysis show that it only contributes to a small part of the total extraction time.

Table 1. Statistics for ConFlEx

Software	# of JBC instr.	# of BIR instr.	BIR time (ms)	Intra-Procedural # of nodes	Intra-Procedural # of edges	Intra-Procedural time (ms)	Inter-Procedural # of nodes	Inter-Procedural # of edges	Inter-Procedural 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

We do not provide comparative data with other extraction tools, such as Soot [22], or Wala [14] because this would demand the implementation of similar extraction rules from their intermediate representations. However, experimental results from Sawja [11] show that it outperforms Soot in all tests w.r.t. the transformation into their respective intermediate representations, and outperforms Wala w.r.t. virtual method call algorithms. Thus, our extraction algorithm clearly benefits from using Sawja and BIR.

4 Correctness of CFG Extraction

This section discusses the correctness proof of the CFG extraction algorithm. Providing a direct proof for our two-phase extraction is cumbersome. Instead, we prove correctness indirectly, using as reference an idealized *direct extraction algorithm*, denoted $\mathtt{m}\mathcal{G}$. The algorithm, defined and proved correct by Amighi [2], is based directly on the semantics of Java bytecode, but assumes an oracle to predict the exceptions that can be thrown by each instruction.

We exploit the idealized algorithm by proving that given a JBC program, the CFG produced by our extraction algorithm ($\mathtt{b}\mathcal{G} \circ \mathtt{BC2BIR}$) structurally simulates the CFG produced by the direct extraction algorithm ($\mathtt{m}\mathcal{G}$). We then reuse an existing result from Gurov *et al.* [10, Th. 36] that structural simulation implies behavioral simulation. By transitivity of simulation we conclude that the behavior induced by the CFG extracted by $\mathtt{b}\mathcal{G} \circ \mathtt{BC2BIR}$ simulates the JVM behavior. Figure 7 summarizes our approach.

The proof of structural simulation is too large to be presented completely in this paper. Instead, we sketch the overall proof, and discuss one case (for the `athrow` instruction) in full detail. For the remaining detailed cases, the reader is referred to the accompanying technical report [3]. Before discussing the proof sketch, we first introduce some terminology and relevant observations.

Preliminaries for the Correctness Proof. The `BC2BIR` transformation may collapse several bytecode instructions into a single BIR instruction. Therefore, we divide bytecode instructions as *producer* instructions, i.e., those that produce at least one BIR instruction in function $\mathtt{BC2BIR}_{instr}$, and *auxiliary* ones, i.e., those that produce none. This division can be deduced from Figure 4 (on page 38).

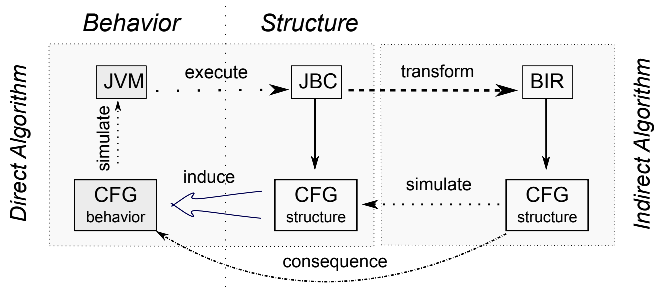

Fig. 7. Schema for CFG extraction and correctness proof

For example, `store` and `invokevirtual` are producer instructions, while `add` and `push` are auxiliary.

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

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

Figure 5 (on page 39) illustrates the partitioning of instructions into segments. Method `odd` has four bytecode (and BIR) segments, as indicated by the coloring. Producer instructions are underlined.

In the definition of the direct extraction algorithm [3], one can observe that all auxiliary instructions give rise to an internal transfer edge only. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path of internal transfer edges with the same size as the number of auxiliary instructions, followed by the edges generated for the producer instruction.

Proof Sketch. Based on observations above, our main theorem states that the method graph extracted using the indirect algorithm weakly simulates (*cf.* [17]) the method graph using the direct algorithm. In the proof, we do not consider the abstract stacks, since only the instructions are relevant to produce the edges.

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

Proof. (Sketch) Let p range over indices in the bytecode instructions array, `pc` over indices in the BIR instructions array, $\circ_m^{p,x,y}$ over control nodes in

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

We define a binary relation R as follows:

$$R \overset{def}{=} \{\ (\circ_m^{p,x,y}, \circ_m^{\mathtt{pc},x,y})\ |\ seg_{JBC}(m,p) = seg_{BIR}(m,\mathtt{pc}) \wedge \mathtt{pc} = min(seg_{BIR}(m,\mathtt{pc}),x,y)\ \}$$

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

The proof proceeds by case analysis on the type of the producer instruction of the bytecode segment $seg_{JBC}(m,p)$. We present one interesting case in full detail; the other cases proceed similarly [3].

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

The direct extraction for the `athrow` instruction produces two edges, with the target node of the second edge depending on whether the exception x is caught within the same method it was raised or not (see [3]):

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

The transformation $\mathtt{BC2BIR}_{instr}$ returns a single instruction. Then, similarly to $\mathtt{m}\mathcal{G}$, the $\mathtt{b}\mathcal{G}$ function produces two edges (see Figure 6):

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

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

We have that $(\circ_m^p, \circ_m^{\mathtt{pc}}) \in R$. The transition $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x}$, is matched by the corresponding weak transition $\circ_m^{\mathtt{pc}} \Longrightarrow \bullet_m^{\mathtt{pc},x}$. Thus obviously also $(\bullet_m^{p,x}, \bullet_m^{\mathtt{pc},x}) \in R$. Next, there are two possibilities for the remaining transitions, depending on whether there is an exception handler for x in p and $\mathtt{pc}$. If there is a handler, then we get $\bullet_m^{p,x} \xrightarrow{handle} \circ_m^q$, $\bullet_m^{\mathtt{pc},x} \overset{handle}{\Longrightarrow} \circ_m^{\mathtt{pc'}}$, and clearly also $(\circ_m^q, \circ_m^{\mathtt{pc'}}) \in R$. If there is no exception handler for x, we get $\bullet_m^{p,x} \xrightarrow{handle} \bullet_m^{p,x,r}$, $\bullet_m^{\mathtt{pc},x} \overset{handle}{\Longrightarrow} \bullet_m^{\mathtt{pc},x,r}$, and also $(\bullet_m^{p,x,r}, \bullet_m^{\mathtt{pc},x,r}) \in R$. This concludes the case. □

5 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.* [18,19] 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.* [15] extend the work of Sinha *et al.* to C++ source code. C++ has the same scheme of `try-catch` and exception propagation as Java, but without the `finally` blocks, or implicit exceptions. This work does not consider the exceptions types. Thus, it heavily over-approximates the possible flows by connecting the control points with explicit `throw` within a `try` block to all its `catch` blocks, and considering that any called method containing a `throw` may terminate exceptionally. Our work consider the exceptions types. Thus, it produces more refined CFGs, and also tells which exceptions can be raised, or propagated from method invocations.

Choi *et al.* [8] use an intermediate representation from the Jalapeño compiler [7] to extract CFGs with exceptional flows. The authors introduce a stackless 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 [16] 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.

6 Conclusion

This paper presents an efficient and sound control-flow graph extraction algorithm from Java bytecode that takes into account exceptional control flow. The extracted CFGs can be used for various control-flow analyses, in particular model checking. The algorithm is precise because it is based on BIR, an intermediate stack-less bytecode representation, which provides precise information about exceptional control-flow, and the result is more compact than the original bytecode.

The algorithm is presented formally as an extraction function. We state and prove its soundness: the behavior of the extracted graphs is shown to over-approximate the behavior of the original programs. To the best of our knowledge, this is the first CFG extraction algorithm that has been proved correct. The proof is non-trivial, relying on several results to obtain a relatively economic correctness argument phrased in terms of structural simulation. We believe that the proposed proof strategy, with the level of detail we provide, paves the ground for a mechanized proof using a standard theorem prover.

The extraction algorithm is implemented as the ConFlEx tool. The experimental results confirm that the algorithm is efficient, and that it produces compact CFGs.

Future Work. The extraction algorithm has been designed with modularity in mind. Currently, we investigate how to relativize the algorithm on interface specifications of program modules in order to support modular control-flow graph extraction. In particular, we target CVPP (see *e.g.* [20,13]), a framework and 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 will study how to adapt the algorithm to various generalizations of the program model, including data and multi-threading [12], and how to customize it for other types of instructions (besides method calls and exceptions).

Acknowledgments. We thank the Celtique team at INRIA Rennes for their clarifications about BIR and Sawja. Amighi and Huisman are partially supported by ERC grant 258405 for the VerCors project.

References

1. ConFlEx, `http://www.csc.kth.se/~pedrodcg/conflex`
2. Amighi, A.: Flow Graph Extraction for Modular Verification of Java Programs. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden (February 2011), `http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/amighi_afshin_11038.pdf`, Ref.: TRITA-CSC-E 2011:038
3. Amighi, A., de Carvalho Gomes, P., Gurov, D., Huisman, M.: Provably correct control-flow graphs from Java programs with exceptions. Tech. rep., KTH Royal Institute of Technology (2012), `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-61188`
4. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA, pp. 324–341 (1996)
5. Barre, N., Demange, D., Hubert, L., Monfort, V., Pichardie, D.: SAWJA API documentation (June 2011), `http://javalib.gforge.inria.fr/doc/sawja-api/sawja-1.3-doc/api/index.html`

6. 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)
7. 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, JAVA 1999, pp. 129–141. ACM, New York (1999)
8. 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 (1999)
9. 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)
10. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
11. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 92–106. Springer, Heidelberg (2011)
12. Huisman, M., Aktug, I., Gurov, D.: Program Models for Compositional Verification. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
13. Huisman, M., Gurov, D.: CVPP: A Tool Set for Compositional Verification of Control–Flow Safety Properties. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 107–121. Springer, Heidelberg (2011)
14. IBM: T.J. Watson Libraries for Analysis (Wala). `http://wala.sourceforge.net/`
15. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. SIGPLAN Not. 42, 3–8 (2007)
16. Jo, J.-W., Chang, B.-M.: Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3043, pp. 106–113. Springer, Heidelberg (2004)
17. Milner, R.: Communicating and mobile systems: the π-calculus, ch. 6, pp. 52–53. Cambridge University Press, New York (1999)
18. Sinha, S., Harrold, M.J.: Criteria for testing exception-handling constructs in Java programs. In: Proceedings of the IEEE International Conference on Software Maintenance, ICSM 1999, pp. 265–276. IEEE Computer Society (1999)
19. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. 26, 849–871 (2000)
20. Soleimanifard, S., Gurov, D., Huisman, M.: ProMoVer: Modular Verification of Temporal Safety Properties. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 366–381. Springer, Heidelberg (2011)
21. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for java. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, pp. 264–280. ACM, New York (2000), `http://doi.acm.org/10.1145/353171.353189`
22. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999), `http://www.sable.mcgill.ca/soot/`