Int J Softw Tools Technol Transfer (2016) 18:653–684
DOI 10.1007/s10009-015-0375-0

CrossMark

REGULAR PAPER

Provably correct control flow graphs from Java bytecode programs with exceptions

Afshin Amighi[1] · **Pedro de Carvalho Gomes**[2] · **Dilian Gurov**[2] · **Marieke Huisman**[1]

Published online: 5 April 2015
© Springer-Verlag Berlin Heidelberg 2015

Abstract We present an algorithm for extracting control flow graphs from Java bytecode that captures normal as well as exceptional control flow. We prove its correctness, in the sense that the behaviour of the extracted control flow graph is a sound over-approximation of the behaviour of the original program. This makes control flow graphs suitable for performing various static analyses, such as model checking of temporal safety properties. Analysing exceptional control flow for Java bytecode is difficult because of the stack-based nature of the language. We therefore develop the extraction in two stages. In the first, we abstract away from the complications arising from exceptional flows, and relativize the extraction on an *oracle* that is able to look into the stack and predict the exceptions that can be raised at each instruction. This *idealized* algorithm provides a specification for concrete extraction algorithms, which have to provide a suitable implementation for the oracle. We prove correctness of the idealized algorithm by means of behavioural simulation. In the second stage, we develop a *concrete* extraction algorithm that consists of 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 provides the information needed to implement the oracle, and since it gives rise to more compact graphs. We show that the behaviour of the control flow graph extracted via the intermediate representation is a sound over-approximation of the behaviour of the graph extracted by the direct, idealized algorithm, and thus of the original program. The concrete extraction algorithm is implemented as the ConFlEx tool. A number of test cases are performed to evaluate the efficiency of the algorithm.

✉ Pedro de Carvalho Gomes
pedrodcg@csc.kth.se

Afshin Amighi
a.amighi@utwente.nl

Dilian Gurov
dilian@csc.kth.se

Marieke Huisman
m.huisman@utwente.nl

[1] University of Twente, Enschede, The Netherlands

[2] KTH Royal Institute of Technology, Stockholm, Sweden

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 have been developed to address this goal, such as various static analyses, model checking and (automated) theorem proving. A major obstacle is presented by the state space of software, which is typically very large or even infinite. Therefore appropriate abstractions are necessary to make the formal analysis tractable. It is important that such abstractions (or *models*) are *sound* w.r.t. the properties of interest about the original program: if a property holds over the abstract model, it should also hold over the original program. To be able to establish formally the soundness of the extracted models one needs a *formalization* of the extraction, as a mapping from programs to models.

A natural abstraction are program models (extracted from program code) that only preserve the information that is relevant for the class of properties at hand. In particular, *control flow graphs* (CFGs) [1] are a widely used abstraction, where

only the control flow information is kept, while all program data is abstracted away. Concretely, in a CFG, nodes represent the control points of a method, and edges represent how instructions shift control between the points.

In the present work, we extract CFGs as program models that are tailored for compositional verification of control-flow-based temporal safety properties in the style of [18,20]. Our definition of CFGs makes two adaptations of the standard notion. First, there are no explicit inter-procedural edges: method calls are represented by labels on the outgoing edges of invocation nodes, and return points are depicted as atomic propositions on sink nodes. Second, the CFGs contain exceptional control nodes, i.e., nodes representing the takeover of control by the Java Virtual Machine (JVM) to handle the exception. The current definition captures method invocations and exceptions only. However, it can easily be extended to observe other types of events, such as lock acquisition or heap manipulation.

For two reasons the analysis of exceptional flows presents a major complication to the sound extraction of CFGs from Java bytecode. First, the stack-based nature of the JVM makes it hard to determine the type of explicitly thrown exceptions, thus making it difficult to statically decide to which handler (if any) control will be transferred. Second, the JVM can raise (implicit) run-time exceptions, such as a `NullPointerException` or an `IndexOutOfBoundsException`; to keep track of where such exceptions can be raised requires much care.

Numerous approaches to the automatic extraction of control flow graphs from program code have previously been presented. However, these are typically not accompanied by any formal correctness argument. We attempt to fill this gap: we present a CFG extraction algorithm for sequential (i.e., single-threaded) Java bytecode (JBC) that captures normal as well as exceptional control flow, and we prove that the extraction algorithm is sound w.r.t. program behaviour, if the latter is viewed as a set of sequential executions (runs). The main challenge here is to come up with a simple formalization of the extraction that allows a relatively straightforward (even if large) soundness proof, to pave the ground for fully formal soundness proofs by means of theorem provers. One complication here derives from the fact that the notion of correctness of the extraction algorithm is *indirect*, in terms of the extracted CFGs being sound models w.r.t. the programs from which they are extracted.

Our extraction algorithm considers all the typical intricacies of (sequential) JBC such as virtual method call resolution, the differences between dynamic and static object types, and exception handling. In particular, it includes explicitly thrown exceptions. Also, it supports a significant subset of the run-time exceptions. This partial support is inherited from the intermediate transformation that our algorithm uses, and the practical aspects of its implementation. In fact, our algorithm can easily be extended to support a wider set of run-time exceptions, as long as the intermediate transformation also does.

We address the problem of sound CFG extraction in *two stages*. In the first, we follow the philosophy of Freund and Mitchell underlying their formalization of the JVM to abstract away from the complications arising from exceptional flows and to relativize the extraction on an *oracle* that is able to look into the stack and predict the exceptions that can be raised at each instruction [13]. The resulting, conceptually simple, idealized algorithm is designed to serve as a *specification* for concrete CFG extraction algorithms, which have to implement the oracle in a suitable fashion. We prove correctness of the algorithm by means of a *simulation* relation between the behaviour of the extracted CFG (in terms of an induced pushdown automaton) and the behaviour of the original JBC program (in terms of its execution on the JVM, following again [13]). The CFGs extracted by the algorithm, however, are rather verbose: in bytecode, all operands are on the stack, thus many instructions for stack manipulation are present, all giving rise to irrelevant edges in the CFG. This affects negatively the efficiency of verification of control flow properties.

To overcome these problems, in the second stage, we develop a concrete extraction algorithm that implements the oracle and at the same time produces more compact CFGs. The algorithm consists of two separate transformations. The first one converts the JBC program into a BIR program. The second transformation defines CFG extraction from BIR. BIR is a stack-less representation of Java bytecode developed by Demange et al. [11]. Thus all instructions (including the explicit `athrow`) are directly connected to their operands, providing the necessary information to implement the oracle. Also, the BIR transformation inserts assertions along the program representation, denoting that a run-time exception can be raised at a given program point. Further, the representation of a program in BIR is smaller than in JBC, because operations are not stack-based, but represented as expression trees. As a result, the extracted CFGs are more compact. The composition of the two transformations constitutes the concrete CFG extraction algorithm from JBC. Its correctness proof uses the correctness of the idealized algorithm. We prove that the CFGs extracted by the idealized algorithm are simulated *structurally* (rather than behaviourally) by the CFGs extracted by the concrete algorithm, which significantly simplifies and shortens the proof. By reusing a previous result from [20] that structural simulation induces behavioural simulation, and by transitivity of simulation, we can deduce behavioural simulation.

The concrete algorithm is implemented as the tool ConFlex. It uses Sawja [19], a library for static analysis of Java bytecode, for the virtual method resolution, and for the BIR transformation. The BIR transformation in Sawja is purely

syntactic. Therefore, we instrumented it to associate types to operands, and to compute the most general type when some operation is performed over operands of different types. Currently SAWJA provides assertions for a subset of the run-time exceptions. CONFLEX supports all the available assertions, and can easily be extended if more are provided. Next, we implemented the CFG extraction algorithm from the BIR representation. It is subdivided into two distinct analyses. The first is intra-procedural, and the CFG of a method is extracted by analysing its instructions only. The second analysis is inter-procedural, and CONFLEX uses a fixed-point computation to determine the flow caused by propagation of uncaught exceptions.

We perform several test cases with CONFLEX to evaluate its efficiency. The experimental results show that the extraction time is linear in the number of instructions of the program. Also, the fixed-point computation of exception propagation is shown to be light-weight in practice, constituting a negligible fraction of the total extraction time. The BIR representation has about one third of the number of instructions of the corresponding JBC program. Thus, it produces more compact graphs than those produced by an implementation of the idealized algorithm that only implements the oracle for the exception analysis.

The results presented here have been partially published in [3] (also presented as part of the Licentiate Thesis of Gomes [14]). The present paper extends that work, expanding on the theoretical underpinnings and providing the details of the formal proofs. Unlike in [3], we present here the indirect algorithm as an instantiation of the idealized algorithm, and provide therefore the formalization and correctness proof of the latter. An earlier version of the idealized algorithm and its correctness proof has been presented as part of Amighi's M.Sc. thesis [2].

Contributions In summary, the contributions of the paper are as follows:

1. A simple *formalization* of CFG extraction from Java bytecode for the verification of temporal safety properties, together with a relatively routine, even though not short soundness proof in terms of a simulation relation relative to a formal JVM specification. The formalization is idealized in the sense that it uses an "oracle" to abstract from the complexities inherent in exceptional control flow, and serves thus as a *specification* for concrete CFG extraction algorithms: any such algorithm (i) has to implement the oracle, and (ii) has only to be shown sound w.r.t. the models extracted by the idealized algorithm.
2. An efficient *instantiation* of the idealized algorithm as a concrete algorithm based on BIR, a well-known intermediate bytecode representation.
3. A *soundness proof* for the concrete algorithm. Even though far from trivial, the proof is greatly simplified by the proof of the idealized algorithm, being performed in terms of structural (i.e., finite state) rather than behavioural (i.e., infinite state) simulation w.r.t. the CFGs extracted by the latter.
4. An *implementation* of the extractor in the shape of the CONFLEX tool. The utility and efficiency of the tool are shown on several test cases.

Organization The remainder of this paper is organized as follows. First, Sect. 2 provides the necessary background definitions for the algorithm and its correctness proof, and exemplifies the application of the current work. Then, Sect. 3 discusses the direct extraction rules for control flow graphs from Java bytecode, while Sect. 4 discusses the indirect extraction rules via BIR, and presents the correctness proof strategy. Section 5 describes the implementation of the indirect algorithm, and presents experimental results. Section 6 discusses implications of the present work. Finally, Sects. 7 and 8 present related work and conclude.

2 Preliminaries

This section briefly reviews the standard formalization of Java bytecode programs of Freund and Mitchell, that forms the basis for our soundness proof, with its corresponding execution environment. It also introduces our model to represent Java bytecode programs, and illustrates how our results are used in a wider context to verify control-flow-based temporal safety properties of Java bytecode programs.

2.1 Java bytecode and the Java virtual machine

Java bytecode is a stack-based executable language. That is, the operands for its instructions are stored on a stack, in contrast to a register-based approach. The Java Virtual Machine (JVM) is a stack-based interpreter that executes Java bytecode programs.

Execution errors of a Java program are reported by the JVM by means of exceptions. Programmers can also explicitly throw exceptions (using instruction `athrow`). Each method can contain multiple exception handlers, which are code blocks executed to recover from an exception. If an exception occurs, and there is no suitable handler in the currently executing method, its execution is terminated abruptly and the JVM continues looking for an appropriate handler in the calling method's context. This process continues until a suitable handler is found, or there are no more calling contexts. In the latter case, execution terminates exceptionally.

The JVM relies upon a module called the *Java bytecode verifier*, that performs type checking and several additional sanity checks over the program code before it starts the execution; e.g., the JVM will not start the execution of a program that contains a method that can terminate by running out of

```
public class EvenOdd {

   public static void main(String[] argv) {
v1        int myarg = Integer.parseInt(argv[1]);
v2,v3     if (argv[0].equals("e"))
v4           even(myarg);
          else
v5           odd(myarg);
v6
   }

    public static boolean odd(int lx) {
v7,v8      if (lx < 0)
v9            throw new ArithmeticException();
v10,v11    else if (lx == 0)
v12           return false;
           else
v13,v14       return even(lx - 1);
    }

     public static boolean even(int lx) {
        try {
v15,v16    if (lx == 0)
v17           return true;
           else
v18,v19       return odd(lx - 1);
        } catch(ArithmeticException e) {
v20,v21       return even(-1 * lx);
        }

     }
}
```

Fig. 1 Example Java source program with control points

instructions, instead of reaching a return instruction. We say a Java bytecode program is *well-formed* if it passes the JVM's bytecode verification.[1]

We use Freund and Mitchell's formal framework for Java bytecode [13]. 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. Subtyping in an environment is denoted by $\Gamma \vdash \tau_1 <: \tau_2$, meaning τ_1 is a subtype of τ_2 in environment Γ. Let METH be a set of method signatures. A method $m \in$ METH is represented in an environment Γ as $\Gamma[m] = \langle B, H \rangle$, where B denotes the body and H the table of exception handlers of method m.

Let ADDR be the set of all valid instruction addresses in Γ, and INST be the Java bytecode instructions set. The body of a JBC method can be considered as a sequence of pairs of addresses and instruction:

$$S' ::= 0 : inst;\ S \qquad S ::= \ \ell : inst;\ S \mid \epsilon$$
$$\ell \in \text{ADDR},\ inst \in \text{INST}$$

The sequence is non-empty, and the address of the first instruction is always zero. $Dom(B) \subset$ ADDR is the set of valid program addresses for method m, and $B[k]$ denotes the instruction at position $k \in Dom(B)$ in the method's body. For convenience, $m[k] = i$ denotes instruction i at location k of method m.

Having EXCP as the set of exceptions, the method's exception table H contains quadruples of the form $\langle b, e, t, x \rangle$, where $b, e, t \in$ ADDR and $x \in$ EXCP. If an exception is thrown by an instruction with index $i \in [b, e)$ and it is from a subtype of x, then $m[t]$ is the first instruction of the corresponding handler. Thus, the instructions between b and e model the `try` block. The instructions starting at t model either the `catch` block that handles the exception x, or a `finally` block, if x is from the special type *any*, defined as an alias of `Throwable`, the super-type of any exception.

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 by:

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

Here, m is the method signature of the active method, p 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 $x \in$ EXCP denotes the exception: in case of an exception, the JVM pushes such a record on the stack.

Example 1 Figure 1 depicts a sample Java source program. It has a single class named `EvenOdd`, containing three methods. The method's control points are annotated in the left column. Figure 2 depicts the same program in Java bytecode. The left column denotes the addresses of the method's instructions, which are also control points in the program execution. The present work covers only the analysis of Java bytecode. However, clearly the JBC representation is much more verbose than the source representation. Therefore, for understandability, we sometimes illustrate definitions using source code programs.

The entry method `main` receives two arguments upon invocation: the first one is a selector between methods `even` and `odd`; the second is the integer to be checked. It invokes two methods from the Java API: `parseInt` and `equals`. The method `odd` potentially throws an `ArithmeticException`. The method `even`, on the other hand, contains an exception handler for such an exception. If an `ArithmeticException` is raised in the interval of control points $[0, 12)$ (`[v15,v19)` in the source), defined by the `try` block, then control is transferred

[1] Requirements available at http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.2.

```
void main(String[]) {
 0: aload_0
 1: iconst_1
 2: aaload
 3: invokestatic
    Integer.parseInt(String)
 6: istore_1
 7: aload_0
 8: iconst_0
 9: aaload
10: ldc            "e"
12: invokevirtual
    String.equals(Object)
15: ifeq           26
18: iload_1
19: invokestatic even(int)
22: pop
23: goto           31
26: iload_1
27: invokestatic  odd(int)
30: pop
31: return
}
```

```
boolean odd(int) {
 0: iload_0
 1: ifge           12
 4: new
    ArithmeticException
 7: dup
 8: invokespecial
    ArithmeticException()
11: athrow
12: iload_0
13: ifne           18
16: iconst_0
17: ireturn
18: iload_0
19: iconst_1
20: isub
21: invokestatic even(int)
24: ireturn
}
```

```
boolean even(int) {
 0: iload_0
 1: ifne         6
 4: iconst_1
 5: ireturn
 6: iload_0
 7: iconst_1
 8: isub
 9: invokestatic odd(int)
12: ireturn
13: astore_1
14: iconst_m1
15: iload_0
16: imul
17: invokestatic even(int)
20: ireturn
Exception table:
<0,12,13,ArithmeticException>
}
```

Fig. 2 Example program in Java bytecode

to the control point 13 (`v20` in source), which is the first instruction defined by the `catch` block.

2.2 Program model

Control flow graphs are abstract models of programs. To define their structure and behaviour, we follow Gurov et al. and use the general notion of *model* [18,20].

Definition 1 (*Model, Initialized Model*) A *model* is a state transition system $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ where S is a set of nodes, L a set of labels, $\rightarrow \subseteq S \times L \times S$ a labelled 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 node $s \in S$. An *initialized model* is a pair $(\mathcal{M}, \mathbb{E})$, where $\mathbb{E} \subseteq S$ is a set of entry nodes.

Control flow graphs are initialized models, where nodes represent the control points of a method, and the edges represent how instructions shift control between the points. In this work we are interested in a specific type of CFGs that abstract from all data, but preserve information about method invocations, and exceptions. Other Java bytecode features are ignored. However, our definitions can easily be extended to capture more features. Also, we do not consider methods from the Java API to be part of the program.

In our CFGs, the nodes contain information about the control points, exceptions and returns. We use the following notation: $\circ_m^{p,r}$ denotes a normal control node, and $\bullet_m^{p,x,r}$ indicates an exceptional control node. The nodes are uniquely identified by their method signature m, position p in the method's instruction array (control address), an optional atomic proposition x (denoting an exception type), and the optional atomic proposition r (denoting a return node).

The edges contain information about invocation instructions. We refer to edges corresponding to such instructions as *visible*, and label them with a method signature. Edges corresponding to other instructions are labelled with ε, and are called *silent*. Invocations to methods from the Java API are also considered silent, although their propagation of exceptions is taken into account.

Example 2 Figure 3 shows the CFG extracted for the program in Example 1. We represent it by means of control points from the Java source, for simplicity. There is one subgraph for each method in the program, and the nodes of each method are tagged with the method's signature. Entry nodes are depicted by incoming edges without source.

There are several exceptional nodes in the CFG (named $\mathtt{e}_1, \mathtt{e}_2, \ldots$) that do not have a corresponding control point in the source code. They represent the configurations in which the control was taken by the JVM, to take care of an exception. The edges from an exceptional node to a normal one denote that there is a handler for the exception in that control point. Exceptional nodes tagged with the atomic proposition r denote the propagation of an exception by the method.

The only visible edges are the ones relative to the invocations of methods `even` and `odd`. Notice that the invocation of `parseInt`, which is a method from the Java API, is considered to be a silent edge. However, the method's signature declares that a `NumberFormatException` (`N.F.E`) is potentially propagated, and this is reflected by the edge to $\mathtt{e}_1$.

Now we define formally CFGs that model sequential programs with procedures and exceptions. Method graphs are the basic building blocks of control flow graphs. They are defined as an instantiation of initialized models as follows.

Definition 2 (*Method Graph*) A *method graph with exceptions* for a method $m \in \text{METH}$ over sets $M \subseteq \text{METH}$ and $E \subseteq \text{EXCP}$ is an initialized model $(\mathcal{M}_m, \mathbb{E}_m)$, where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ with V_m being the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$ the set of labels,

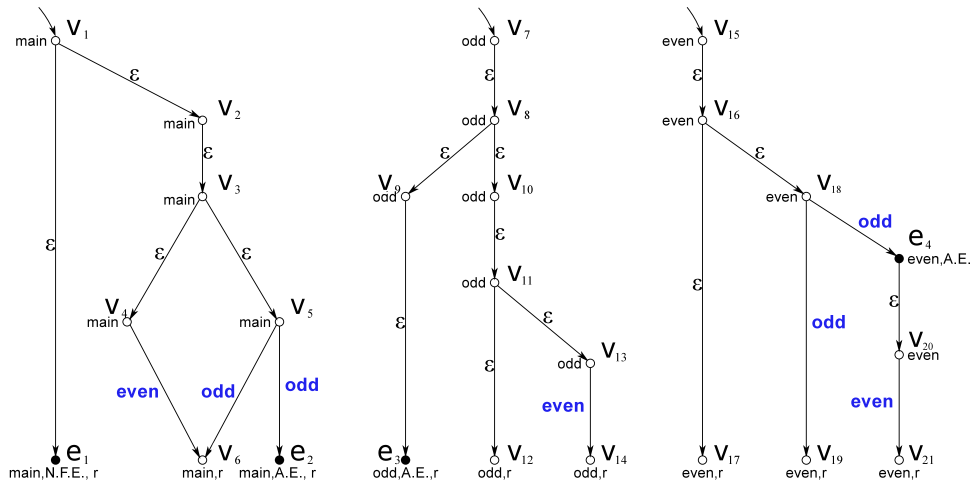

Fig. 3 Control flow graph for the example Java source program

$A_m = \{m, r\} \cup E$, $m \in \lambda_m(v)$ 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 signature of the method 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 CFG is essentially a collection of method graphs. Also, every control flow graph $\mathcal{G}$ is equipped with an interface $I = (I^+, I^-, I^e)$, denoted $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$, where $I^+, I^- \subseteq$ METH are the set of *provided*, and (externally) *required* methods, respectively. We say a CFG is *closed* if all the required methods are also provided; we say it is *open* otherwise. $I^e \subseteq I^+ \times \mathrm{E}$ is the finite set of potentially propagated exceptions by each provided method. The *composition* of CFGs is defined as the disjoint union $\uplus$ of their method graphs. We should stress here that in the composition, having the definition of I^e as a pair assists us to track the method that propagates the exception. The composition of two interfaces is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, (I_1^- \cup I_2^-) \backslash (I_1^+ \cup I_2^+), I_1^e \cup I_2^e)$.

Example 3 The method graph of `odd` is the central subgraph in Fig. 3, and its interface is ({`odd`}, {`even`}, {(`odd`, `ArithmeticException`)}). The composed CFG of the program is the disjoint union of all method graphs, as in Fig. 3. Its interface is ({`main`, `odd`, `even`}, {}, { (`main`, `NumberFormatException`), (`main`, `Arithmetic-Exception`), (`odd`, `ArithmeticException`)}).

The operational semantics of CFGs, referred to here as CFG *behaviour*, is defined also as an instance of an initialized model. A CFG induces a behaviour in terms of a push-down automaton, modeling the JVM call stack. Intuitively, the CFG behaviour is an abstraction of the JVM behaviour, where the activation records are mapped to control nodes, and the only information preserved is the method signature, program point, and a potential exception. The behaviour of CFGs is defined as follows.

Definition 3 (*CFG Behaviour*) Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed flow graph with exceptions such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The *behaviour* of $\mathcal{G}$ is described by the initialized model $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$ such that:

- $S_b \in V \times V^*$, i.e., states are pairs of control node and stack of control nodes,
- $L_b = \{\tau\} \cup L_b^C \cup L_b^X$ where $L_b^C = \{m_1\ l\ m_2 \mid l \in \{call, ret, xret\}, m_1, m_2 \in I^+\}$ (the set of call and return labels) and $L_b^X = \{l\ x \mid l \in \{\text{throw, catch}\}, x \in \text{EXCP}\}$ (the set of exceptional transition labels).
- $A_b = A$
- $\lambda_b((v, \sigma)) = \lambda(v)$
- $\rightarrow_b \subset S_b \times L_b \times S_b$ is the set of transitions defined by the following rules:

$$[transfer] \quad (v,\sigma) \xrightarrow{\tau}_b (v',\sigma) \quad \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ r \notin \lambda(v),\ \lambda(v) \cap \text{EXCP} = \lambda(v') \cap \text{EXCP} = \emptyset$$

$$[call] \quad (v_1,\sigma) \xrightarrow{m_1\ call\ m_2}_b (v_2, v_1 \cdot \sigma) \quad \text{if } \{m_1, m_2\} \subseteq I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1',\ m_2 \in \lambda(v_2),\ v_2 \in \mathbb{E},\ r \notin \lambda(v_1),\ \lambda(v_1) \cap \text{EXCP} = \lambda(v_2) \cap \text{EXCP} = \emptyset$$

$$[return] \quad (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ ret\ m_1}_b (v_1', \sigma) \quad \text{if } \{m_1, m_2\} \subseteq I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1',\ \{m_2, r\} \subset \lambda(v_2),\ m_1 \in \lambda(v_1),\ m_1 \in \lambda(v_1'),\ \lambda(v_1') \cap \text{EXCP} = \emptyset$$

$$[xreturn] \quad (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ xret\ m_1}_b (v_1', \sigma) \quad \text{if } \{m_1, m_2\} \subseteq I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1'\ x \in \text{EXCP},\ x \notin \lambda(v_1),\ m_1 \in \lambda(v_1),\ \{m_2, x, r\} \subseteq \lambda(v_2),\ \{m_1, x\} \subseteq \lambda(v_1')$$

$$[throw] \quad (v,\sigma) \xrightarrow{throw\ x}_b (v',\sigma) \quad \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ r \notin \lambda(v),\ x \in \text{EXCP},\ x \in \lambda(v')$$

$$[catch] \quad (v,\sigma) \xrightarrow{catch\ x}_b (v',\sigma) \quad \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ r \notin \lambda(v),\ x \in \text{EXCP},\ x \in \lambda(v),\ \lambda(v') \cap \text{EXCP} = \emptyset,$$

The set of entry states is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$, where ϵ is the empty sequence.

Intuitively, τ-transitions model transfer of control between nodes. A *throw*-transition models the raise of an exception,

and a *catch*-transition models the transfer of control to an exception handler. In these cases, the stack is not changed. A *call*-transition models a method invocation: the calling node is pushed onto the stack, and control is transferred to the entry node of the callee method. A *return*-transition models the normal termination of a method: the calling node is popped from the stack, and control is transferred to the successor normal control node. An *xreturn*-transition models the abortion of a method execution by an uncaught exception x, and its propagation: the calling node is popped from the stack, and control is transferred to the successor exceptional node tagged with x.

Our definition is based on Huisman et al. [20, Definition 8], but there is a difference in how the method calls and returns are modeled. In the original definition, the set of all the return nodes, either normal or exceptional, is pushed onto the stack upon a method call, and then the return transitions are defined for the set of nodes in the stack. In our definition, only the calling node is pushed onto the stack, and we define transitions based on the called method's return points. Also, we partition the return transitions into normal and exceptional. We introduce such changes to mimic closer the JVM behaviour, and to stress the propagation of exceptions. However, notice that both definitions are *equivalent*, since the pushed calling node can be seen as a place-holder for the set of all return nodes.

Example 4 Consider the CFG in Fig. 3. Following is an example run through the (infinite-state) behaviour induced by the CFG:

$$
\begin{aligned}
&(v_1, \epsilon) \xrightarrow{\tau}_b (v_2, \epsilon) \xrightarrow{\tau}_b (v_3, \epsilon) \xrightarrow{\tau}_b (v_4, \epsilon) \\
&\quad \xrightarrow{\texttt{main}\ call\ \texttt{even}}_b (v_{15}, v_4) \xrightarrow{\tau}_b (v_{16}, v_4) \xrightarrow{\tau}_b \\
&(v_{18}, v_4) \xrightarrow{\texttt{even}\ call\ \texttt{odd}}_b (v_7, v_{18} \cdot v_4) \\
&\quad \xrightarrow{\tau}_b (v_8, v_{18} \cdot v_4) \xrightarrow{\tau}_b (v_9, v_{18} \cdot v_4) \xrightarrow{throw\ \texttt{A.E.}}_b \\
&(e_3, v_{18} \cdot v_4) \xrightarrow{\texttt{odd}\ xreturn\ \texttt{even}}_b (e_4, v_4) \\
&\quad \xrightarrow{catch\ \texttt{A.E.}}_b (v_{20}, v_4) \xrightarrow{\tau}_b \quad \cdots
\end{aligned}
$$

This sample represents an execution starting in the entry control point of the `main` method, next invoking `even`, and then `odd`. An `ArithmeticException` (`A.E.`) is thrown, but not caught, during the execution of `odd`, and causes the method to terminate. The exception is propagated to the calling method `even`, which catches it, and the execution proceeds.

2.3 Verification of control-flow-based temporal safety properties

The current work describes how to extract CFGs that preserve sequences of method calls and exceptions. We illustrate the utility of the extracted CFGs by briefly describing the application and context that motivated the algorithms and tool presented in this paper. However, we should stress that the extracted CFGs are also useful for other types of program analyses, such as [4,17].

Gurov et al. developed a technique for the verification of control-flow-based temporal safety properties, using CFGs as a program model [18]. The properties are specified in Simulation Logic, which is the (safety) fragment of the modal μ-calculus [26] without diamond modalities and least fixed points. The correctness of the verification results is therefore only guaranteed for models that are *sound* w.r.t. this class of properties. The framework has been extended to also support the more intuitive (linear-time) temporal logic Weak LTL, which is the (safety) fragment of LTL [30] that uses the weak version of the until temporal operator. The technique is tailored for compositional verification, but can also be used in a non-compositional setting.

With the technique of [18] one can thus verify properties over sequences of method invocations and exceptions, using the CFGs extracted with the algorithm presented here. Examples of useful properties are: "Exception X will only be caught by a certain method M", "If exception X is thrown, the first invoked method must be the state-restoring method M", and "Exception X is always caught within the method that raised it, or by its caller method."

The verification technique is implemented as CVPP [21], a tool set for the (compositional) verification of JBC programs w.r.t. temporal safety properties of sequences of method invocations, and exceptions. CVPP is wrapped by ProMoVer [35,36], a tool that automatically encapsulates the verification steps in CVPP. Below we illustrate how our extraction algorithm fits in CVPP for the (non-compositional) verification of a control-flow-based temporal safety property.

When verifying a Java bytecode program, the first step is the extraction of its CFG. Next, the CFG is used to construct a push-down system (PDS) that represents the induced CFG behaviour, following the operational semantics from Definition 3. Finally, CVPP verifies the temporal property of interest against the PDS by using a standard PDS model checker [24,31].

Example 5 Suppose we wish to verify whether the program in Fig. 2 will never abort because of an uncaught `ArithmeticException`. The following formula in Weak LTL expresses this property, where G is the temporal operator *globally* (or *always*). The formula essentially states that program control never reaches a return point of method `main` tagged with the given exception type:

$$\phi_1 = \mathrm{G}\,\neg(\texttt{main} \wedge r \wedge \texttt{ArithmeticException})$$

CVPP extracts the CFG from the program, and creates a PDS that represents its behaviour. Both the PDS and ϕ_1 are inputs

to the PDS model checker. The checker establishes that ϕ_1 does not hold for the CFG behaviour. A *counter-example* produced by the checker exhibits a run that violates ϕ_1: initially `main` invokes `odd`, then `odd` throws, but does not catch, an `ArithmeticException`, the exception is then propagated to `main`, and finally causes its exceptional termination.

Now suppose we wish to check whether whenever `even` is the first method invoked by `main`, then the program cannot abort because an `ArithmeticException` is not caught. The following Weak LTL formula expresses this property, where W is the standard temporal operator *weak until*:

$$\phi_2 = (\texttt{main}\ \mathrm{W}\ \texttt{even}) \rightarrow \mathrm{G}\neg(\texttt{main} \wedge r \wedge \texttt{ArithmeticException})$$

Again, CVPP feeds ϕ_2 and the PDS into the model checker, which shows that ϕ_2 holds for the behaviour induced by the CFG from the program in Example 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. The rules are defined purely syntactically, based on the method's instructions. However, they are justified intuitively by the instructions' operational semantics. The algorithm is idealized, and relies on an oracle to provide information about exceptions.

In the following we classify the set of the instructions based on their operational semantics, then we introduce the direct extraction algorithm and finally we prove the soundness of the proposed algorithm.

3.1 JBC instructions

The essence of any CFG extraction algorithm is the analysis of all possible transitions between any two control points of the program. Thus, our algorithm is based on the operational semantics of the Java bytecode, as defined in [13]. We introduce two simplifications. The first is the assumption of an oracle to provide information about exceptions. It lists which exceptions are potentially raised by the execution of each instruction type. Also, the instruction `athrow` (explicit exception throw) does not have an argument; instead the exception is determined at run-time by the top of the stack. Our algorithm replaces `athrow` with `throw X`, and uses the oracle to list the set `X` of possible exception types. Second, the instructions `jsr q` and `ret r`, for the invocation and return of subroutines, are not considered because they are deprecated since JBC version 1.6 [27].

We extend the set of the method signatures with a special signature named M_{jvm} to denote the method calling the `main` method and then group the instructions with common behaviour w.r.t. the control flow into disjoint subsets. The JBC instructions set contains more than 200 instructions, and this classification enables us to have a manageable list of extraction rules. The behaviour of subsets is defined by the operational semantics, which is also used to justify our extraction rules. Figure 4 presents the partitioned sets, an intuitive description of the semantics, and examples of instructions that have a common control flow behaviour.

Now we introduce some auxiliary functions, necessary to define the operational semantics and formally group the instructions. The function $succ(p)$ takes a control point and based on the instruction at this control point yields next control point. To extract the branch address of CNDINST and JMPINST we define $jmp(i)$ which accepts an instruction of types CNDINST and JMPINST and returns the branch address. For example, $jmp(i)$, returns `q` if $i =$ `ifeq q` or $i =$ `goto q` where `q` is the branch address. The function *cond* receives an activation record, and evaluates the condition of the instruction in $m[p]$. It is defined as follows for instruction $i =$ `ifeq q` and can be defined for the rest of the conditional instructions similarly.

$$cond(\langle m, p, f, v_1 \cdot s, z\rangle) = \begin{cases} \mathbf{tt} \text{ if } (m[p] = \texttt{ifeq q} \wedge v_1 = 0) \\ \mathbf{ff} \text{ if } (m[p] = \texttt{ifeq q} \wedge v_1 \neq 0) \end{cases}$$

The function $k^{p,x}_{\Gamma[m]}$ is a lookup function to search top-down the exception table of a method implementation $\Gamma[m]$ for a *suitable handler* for an exception x at location p. It returns zero if no handler is found, or the position t of the first suitable handler. It is defined formally as follows:

$$k^{p,x}_{\Gamma[m]} = \begin{cases} t & \text{if } \exists\, b, e, x'.\ \langle b, e, t, x'\rangle \in H_{\Gamma[m]} \ \wedge\ p \in [b, e) \\ & \quad \wedge\ \Gamma \vdash x <: x' \ \wedge\ t \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Subset	Description	Examples
CMPINST	Computational instructions	`nop, push v, pop`
CNDINST	Conditional instructions	`ifeq q`
JMPINST	Jump instructions	`goto q`
THRINST	Explicit exception throw	`throw X`
XMPINST	Potentially can raise exceptions	`div, getfield f, new, newarray`
INVINST	Method invocations	`invokevirtual (o,m), invokespecial (o,m)`
RETINST	Normal return instructions	`return`

Fig. 4 Grouping of bytecode instructions with common control flow behaviour

$$[\textsc{Cmp}]\ \frac{m[p]=i\in\textsc{CmpInst}}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{\tau}(\langle m,succ(p),f',s',z'\rangle\cdot A;h')}$$

$$[\textsc{Cnd}]\ \frac{m[p]=i\in\textsc{CndInst}\quad cond(\langle m,p,f,s,z\rangle)=true}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{\tau}(\langle m,jmp(i),f,s',z\rangle\cdot A;h)}$$

$$[\textsc{Cnd}]\ \frac{m[p]=i\in\textsc{CndInst}\quad cond(\langle m,p,f,s,z\rangle)=false}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{\tau}(\langle m,succ(p),f,s',z\rangle\cdot A;h)}$$

$$[\textsc{Jmp}]\ \frac{m[p]=i\in\textsc{JmpInst}}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{\tau}(\langle m,jmp(i),f,s,z\rangle\cdot A;h)}$$

$$[\textsc{Thr}]\ \frac{m[p]=i\in\textsc{ThrInst}}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{throw\ x}(\langle x\rangle_{\mathsf{exc}}\cdot\langle m,p,f,s,z\rangle\cdot A;h)}$$

$$[\textsc{Xmp}]\ \frac{m[p]=i\in\textsc{XmpInst}\quad \nu(\langle m,p,f,s,z\rangle,h)=x}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{throw\ x}(\langle x\rangle_{\mathsf{exc}}\cdot\langle m,p,f,s',z\rangle\cdot A;h')}$$

$$[\textsc{Exc}]\ \frac{k^{p,x}_{\Gamma[m]}=t\ \wedge\ t\neq 0}{(\langle x\rangle_{\mathsf{exc}}\cdot\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{catch\ x}(\langle m,t,f,s,z\rangle\cdot A;h')}$$

$$[\textsc{Exc}]\ \frac{k^{p,x}_{\Gamma[m]}=0}{(\langle x\rangle_{\mathsf{exc}}\cdot\langle m,p,f,s,z\rangle\cdot\epsilon;h)\xrightarrow{m\ xret\ M_{jvm}}(\langle x\rangle_{\mathsf{exc}}\cdot\epsilon;h')}$$

$$[\textsc{Exc}]\ \frac{k^{p,x}_{\Gamma[m]}=0}{(\langle x\rangle_{\mathsf{exc}}\cdot\langle n,q,f,s,z\rangle\cdot\langle m,p,f',s',z'\rangle\cdot A;h)\xrightarrow{n\ xret\ m}(\langle x\rangle_{\mathsf{exc}}\cdot\langle m,p,f',s',z'\rangle\cdot A;h')}$$

$$[\textsc{Xmp}]\ \frac{m[p]=i\in\textsc{XmpInst}\quad \nu(\langle m,p,f,s,z\rangle,h)=undef}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{\tau}(\langle m,succ(p),f',s',z'\rangle\cdot A;h')}$$

$$[\textsc{Inv}]\ \frac{m[p]=i\in\textsc{InvInst}\quad \nu(\langle m,p,f,s,z\rangle,h)\neq\mathtt{N.P.E.}}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{m\ call\ n}(\langle n,0,f',\epsilon,z'\rangle\cdot\langle m,p,f,s,z\rangle\cdot A;h')}$$

$$[\textsc{Inv}]\ \frac{m[p]=i\in\textsc{InvInst}\quad \nu(\langle m,p,f,s,z\rangle,h)=\mathtt{N.P.E.}}{(\langle m,p,f,s,z\rangle\cdot A;h)\xrightarrow{throw\ \mathtt{N.P.E.}}(\langle\mathtt{N.P.E.}\rangle_{\mathsf{exc}}\cdot\langle m,p,f,s,z\rangle\cdot A;h')}$$

$$[\textsc{Ret}]\ \frac{m[p]=i\in\textsc{RetInst}}{(\langle m,p,f,s,z\rangle\cdot\langle n,q,f',s',z'\rangle\cdot A;h)\xrightarrow{m\ ret\ n}(\langle n,succ(q),f',s'',z'\rangle\cdot A;h)}$$

$$[\textsc{Ret}]\ \frac{m[p]=i\in\textsc{RetInst}}{(\langle m,p,f,s,z\rangle\cdot\epsilon;h)\xrightarrow{m\ ret\ M_{jvm}}(\epsilon;h)}$$

Fig. 5 JBC instructions operational semantics

The partial function ν receives an activation record $\langle m, p, f, v\cdot s, z\rangle$ and a heap h, and yields the possible runtime exceptions that an instruction $m[p]$ might raise, if any. We present the definition of this function for three sample of instructions; for all remaining ones it can be defined similarly.

to configuration c' labelled by a behavioural label $l \in L_b$ as defined in Definition 3. Inside activation records, we use f', s' and z' to denote that f, s and z are updated in the transition.

$$\nu(\langle m, p, f, v\cdot s, z\rangle, h) = \begin{cases} \mathtt{NullPointerException} & \text{if } m[p] = \mathtt{getfield\ f} \wedge v = null \\ \mathtt{NegativeArraySizeException} & \text{if } m[p] = \mathtt{newarray\ a} \wedge v < 0 \\ \mathtt{ExceptionInInitializerError} & \text{if } m[p] = \mathtt{new} \end{cases}$$

We should stress here that the operational semantics for `new` in [13] does not define any exception for the instruction. But, actually the `new` instruction can potentially raise `ExceptionInInitializerError`. This will be elaborated more in 4.1.

Figure 5 presents the operational semantics for each subset of the JBC instructions. The rules are presented in the form:

$$[\textsc{Name}]\ \frac{P_0 \quad \cdots \quad P_n}{c \xrightarrow{l} c'}$$

where Name is the name of the rule, $\{P_0, \ldots, P_n\}$ is a set of premises, and $c \xrightarrow{l} c'$ is a transition from configuration c

Each group in Fig. 4 corresponds to one of the rules defined in Fig. 5, except for the rules named Exc, which are called JVM's *internal transitions*. These refer to a set of the transitions that are not controlled by program instructions. In particular when the program raises an exception, there is no instruction in the JBC instructions set to handle this exception. Instead, the JVM takes control of the execution, and looks for a suitable handler for the exception. In case a handler is found, the JVM sets control to this handler, and the program continues with the execution. Otherwise, the method is aborted, and the exception is propagated to the caller method.

3.2 The extraction algorithm from JBC

We now present how to extract a CFG, as defined in Sect. 2.2, from a given JBC program. First, we define the elements of a CFG from a program's method. The nodes of a method's CFG are tagged with an address and a method signature. Based on Definition 2, to construct the nodes we have to specify V_m, A_m, λ_m, and $\mathbb{E}_m$. A node $v \in V_m$ is uniquely identified by its control point $p \in$ ADDR and its atomic propositions, as tagged by the function $\lambda_m(v)$. The method signature is the default tag for all the method's control nodes. If $m[p] \in$ RETINST then the node is tagged with r. If the node is an exceptional node then it is tagged with the exception type $x \in E$. If $p = 0$ then the node will be a member of $\mathbb{E}_m$.

Given a JBC program, our algorithm extracts the CFG for each class defined in the program. The CFG of a class is the CFG composed from all the methods defined in the class. Thus, to build the CFG for the program, we extract the CFG for each method. The CFG extraction rules for method m in environment Γ use the implementation of the method: $\Gamma[m] = \langle B, H\rangle$. For each instruction in B (body of $\Gamma[m]$), the rules build a set of labelled edges connecting control nodes. In addition, the algorithm performs an inter-procedural analysis to establish the call-return relation between methods.

Definition 4 (*CFG Extraction from JBC*) The *instruction-wise* extraction function $\mathcal{G}_{\text{JBC}} : (\text{METH} \times \text{ADDR} \times \text{JBCINST}) \rightarrow \mathcal{P}(V_m \times L_m \times V_m)$ is defined by the rules in Fig. 6. The method graph for m is defined as $\mathcal{G}_{\text{JBC}}(m) = \bigcup_{(p,i)\in B} \mathcal{G}_{\text{JBC}}^{m,p,i}$. The control flow graph for the program is defined as $\mathcal{G}_{\text{JBC}}(\Gamma) = \bigcup_{\{m \mid \Gamma[m]\in\Gamma\}} \mathcal{G}_{\text{JBC}}(m)$.

We start the explanation of the construction rules with the auxiliary functions. There are four main auxiliary functions to construct edges: for handlers, exception raising, exception propagation, and virtual method calls.

Auxiliary functions The function $\mathcal{H}_m^{x,p,l}$ constructs the edges corresponding to the raising and handling of an exception. It queries the function $k_{\Gamma[m]}^{p,x}$ for a suitable handler for an exception of type x at position p. If there is a handler, it returns a pair of edges: one labelled with parameter l from a normal to an exceptional node (tagged with x), and one labelled with ε from the exceptional node to a normal node, denoting control transfer to the handler. Otherwise it produces an edge labelled with l to an exceptional return node, denoting the propagation of the exception to the caller. The parameter l is instantiated with a method signature if the exception is propagated, or with ε if the exception is raised within the method.

The auxiliary function $\mathcal{E}$ uses $\mathcal{H}$ to compute exceptional edges for all exceptions that can potentially be raised by a given instruction. The function $\mathcal{X}$: XMPINST $\rightarrow \mathcal{P}$(EXCP) returns the set of run-time exceptions that an instruction may raise. The `throw` instruction is handled similarly, where X is the set of possible exceptions. Both rely on an oracle to list the set of exceptions. Function $\mathcal{N}_m^{p,n}$ generates the set of edges to handle all uncaught exceptions from the possible callee n. Notice that it performs an inter-procedural analysis, since it evaluates the exceptional return nodes from the callee method's CFG.

To extract edges for method invocations, the auxiliary function Rec_Γ^i yields the set of possible method signatures of a method call in environment Γ. The receiver object for `invokevirtual` is determined by late binding. For this, the virtual method call resolution function res_Γ^α is employed, where α is a parameter denoting an external standard static analysis to resolve the call. We use n_T to indicate method n from class T. For example, Rapid Type Analysis (RTA) [5] returns the set of subtypes of the caller's static type which are

$$\mathcal{H}_m^{x,p,l} = \begin{cases} \{(\circ_m^p, l, \bullet_m^{p,x}), (\bullet_m^{p,x}, \varepsilon, \circ_m^t)\} & \text{if } k_{\Gamma[m]}^{p,x} = t \neq 0 \\ \{(\circ_m^p, l, \bullet_m^{p,x,r})\} & \text{if } k_{\Gamma[m]}^{p,x} = 0 \end{cases}$$

$$\mathcal{E}_m^p = \bigcup_{x\in\mathcal{X}(m[p])} \mathcal{H}_m^{x,p,\varepsilon}$$

$$\mathcal{N}_m^{p,n} = \bigcup_{\{x \mid \bullet_n^{q,x,r}\in V_n\}} \mathcal{H}_m^{x,p,n}$$

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

$$\mathcal{G}_{\text{JBC}}^{m,p,i} = \begin{cases} \{(\circ_m^p, \varepsilon, \circ_m^{succ(p)})\} & \text{if } i \in \text{CMPINST} \\ \{(\circ_m^p, \varepsilon, \circ_m^{jmp(i)})\} & \text{if } i \in \text{JMPINST} \\ \{(\circ_m^p, \varepsilon, \circ_m^{succ(p)}), (\circ_m^p, \varepsilon, \circ_m^{jmp(i)})\} & \text{if } i \in \text{CNDINST} \\ \{(\circ_m^p, \varepsilon, \circ_m^{succ(p)})\} \cup \mathcal{E}_m^p & \text{if } i \in \text{XMPINST} \\ \bigcup_{x\in X} \mathcal{H}_m^{x,p,\varepsilon} & \text{if } i \in \text{THRINST} \\ \bigcup_{n\in Rec_\Gamma^i} \{(\circ_m^p, n, \circ_m^{succ(p)})\} \cup \mathcal{H}_m^{\texttt{N.P.E.},p,\varepsilon} \cup \mathcal{N}_m^{p,n} & \text{if } i \in \text{INVINST} \\ \emptyset & \text{if } i \in \text{RETINST} \end{cases}$$

Fig. 6 CFG construction rules

instantiated in the program (created by a `new` instruction). So, for $\alpha = RTA$, the result of the resolution for object o and method n in environment Γ will be:

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

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 in τ, i.e., τ is a subtype of o's static type and method n is defined in class τ.

Construction rules For simple computational instructions, a direct edge to the next control address is established. For jump instructions, an edge to the jump address ($jmp(i)$, for instruction i) is generated. For conditional instructions, edges to the next control address and to the address specified for the jump 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 $\mathcal{H}_m^{x,p,l}$. Required edges for instruction `throw` can simply be produced by using function $\mathcal{H}$ for all the exceptions in the over-approximated instruction parameter X.

Given the set of possible receivers, required edges are generated for each possible receiver. For each call, if the method's execution terminates normally, control will be given back to the next instruction of the caller. If the method terminates with an uncaught exception, the caller has to handle this propagated exception. The CFG extraction rule for method invocation produces edges for both `NullPointerException` (`N.P.E.`) (in case of a null receiver object exception) and for all propagated exceptions. To generate exceptional edges for `N.P.E.`, $\mathcal{H}_m^{\texttt{N.P.E.},p,\varepsilon}$ is employed, and $\mathcal{N}_m^{p,n}$ generates the set of edges to handle all uncaught exceptions propagated by any possible callee n.

In all the rules, if the target node points to an instruction $i \in$ RETINST then the node will be tagged with r. To keep the presentation of the rules reasonably simple we do not show the target node checks in Fig. 6. Thus the rule for $i \in$ RETINST does not generate any edges in the CFG.

3.3 Soundness of CFG extraction

To show soundness of the extraction algorithm w.r.t. temporal safety properties over sequences of method calls in the possible presence of exceptions, we show that the executions of the extracted CFG of a program P can match the labelled transitions[2] of any execution of P, i.e., it over-approximates the behaviour of P. To this end, we define a mapping θ that abstracts JVM configurations into CFG behavioural configurations, and use this abstraction to prove that the behaviour of the CFG *simulates* the behaviour of the program, in the standard sense of simulation between labelled transition systems [29].

[2] Except configurations $(\epsilon; h)$, since CFG configurations always have a current control point.

Definition 5 (*VM State Abstraction* θ) Let CONF be the set of JVM execution configurations of program P, V the set of control nodes from the extracted control flow graph $\mathcal{G}_{\text{JBC}}$, and S_b the set of CFG behavioural configurations of $b(\mathcal{G}_{\text{JBC}})$. First, define the mapping γ abstracting any sequence of activation records ω into a sequence of nodes in $\mathcal{G}_{\text{JBC}}$, inductively as follows:

$$\gamma(\omega) = \begin{cases} \epsilon & \text{if } \omega = \epsilon \\ \circ_m^p \cdot \gamma(\omega') & \text{if } \omega = \langle m, p, f, s, z\rangle \cdot \omega' \wedge m[p] \notin \text{RETINST} \\ \circ_m^{p,r} \cdot \gamma(\omega') & \text{if } \omega = \langle m, p, f, s, z\rangle \cdot \omega' \wedge m[p] \in \text{RETINST} \\ \bullet_m^{p,x} \cdot \gamma(\omega') & \text{if } \omega = \langle x\rangle_{\texttt{exc}} \cdot \langle m, p, f, s, z\rangle \cdot \omega' \wedge k_{\Gamma[m]}^{p,x} \neq 0 \\ \bullet_m^{p,x,r} \cdot \gamma(\omega') & \text{if } \omega = \langle x\rangle_{\texttt{exc}} \cdot \langle m, p, f, s, z\rangle \cdot \omega' \wedge k_{\Gamma[m]}^{p,x} = 0 \end{cases}$$

Then, given a configuration $(\omega; h) \in$ CONF with $\omega \neq \epsilon$, define its abstraction as the state $\theta(\omega; h) = (head(\gamma(\omega)), tail(\gamma(\omega))) \in \text{S}_b$.

We now enunciate the soundness theorem and present two related cases of the proof; the complete proof can be found in Appendix 9.

Theorem 1 (CFG Soundness) *Let P_{jbc} be a well-formed Java bytecode program modeled by the environment Γ. The behaviour of the extracted flow graph $\mathcal{G}_{\text{JBC}}(\Gamma)$ simulates the execution of P_{jbc}.*

Proof We prove simulation between the execution of P_{jbc} and the behaviour of the extracted CFG, i.e. $b(\mathcal{G}_{\text{JBC}})$. In general, this requires to show that every initial configuration of P_{jbc} is simulated by some initial configuration of $b(\mathcal{G}_{\text{JBC}})$. In our case there is a single initial program configuration:

$$c_{init} = (\langle main, 0, f, \epsilon, z\rangle; h)$$

The required simulation can be established by exhibiting a concrete *simulation relation* between JVM configurations and CFG behavioural configurations that relates the initial P_{jbc}-configuration c_{init} to an initial $b(\mathcal{G}_{\text{JBC}})$-state.

We prove that the abstraction function θ defined above, viewed as a relation, is such a simulation relation. First, observe that $\theta(c_{init}) = (v^0_{main}, \epsilon)$ is an initial state of $b(\mathcal{G}_{\text{JBC}})$, see Definition 3. Then, we prove that for any JVM configuration c and any transition $c \xrightarrow{l} c'$ in the execution of P_{jbc}, there is a matching transition $\theta(c) \xrightarrow{l} \theta(c')$ in $b(\mathcal{G}_{\text{JBC}})$. We show this by case analysis on the top frame $\langle m, p, f, s, z\rangle$ or $\langle x\rangle_{\texttt{exc}}$, and in the former case also on the type of the instruction $m[p]$, i.e., the current JBC instruction. For each case we deduce the possible labelled transitions $c \xrightarrow{l} c'$ from c

as induced by the JBC operational semantics rules shown on Fig. 5. Next, we use the CFG construction rules from Fig. 6 to determine the nodes and edges extracted for instruction $m[p]$, and the definition of CFG behaviour (Definition 3) to establish $\theta(c) \xrightarrow{l} \theta(c')$.

We now present the two related cases of raising an exception and then handling the exception. The remaining cases can be found in Appendix 9.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $m[p] \in$ ThrInst. The single instruction in the set is `throw X`. Let X be the set containing the static type of the exception being thrown, and all of its subtypes. Applying the abstraction function on configuration c we obtain: $\theta(c) = (\circ_m^p, \sigma)$ where σ denotes $\gamma(A)$. The operational semantics of `throw X` (see Fig. 5, rule [Thr]) defines that for every exception $x \in X$ there is a next configuration c' such that: $c \xrightarrow{throw\ x} c'$ where $c' = (\langle x\rangle_{\mathrm{exc}} \cdot \langle m, p, f, s, z\rangle \cdot A; h)$. Then there are two sub-cases:

1. Method m has a handler for x (i.e. $k^{p,x}_{\Gamma[m]} = t \ \wedge\ t \neq 0$). From the definition of the CFG construction function $\mathcal{G}_{\mathrm{JBC}}$ (see Fig. 6, case $i =$ `throw X`), we have: $(\circ_m^p, \varepsilon, \bullet_m^{p,x}) \in \mathcal{G}_{\mathrm{JBC}}^{m,p,\mathtt{throw\ X}}$.
   By the definition of CFG behaviour (see Definition 3, rule [*throw*]), this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{throw\ x}_b (\bullet_m^{p,x}, \sigma)$.
   Now, since $k^{p,x}_{\Gamma[m]} \neq 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet_m^{p,x}, \sigma)$ and therefore: $\theta(c) \xrightarrow{throw\ x} \theta(c')$.
2. There is no handler in m for x (i.e. $k^{p,x}_{\Gamma[m]} = 0$). From the CFG extraction function we obtain: $(\circ_m^p, \varepsilon, \bullet_m^{p,x,r}) \in \mathcal{G}_{\mathrm{JBC}}^{m,p,\mathtt{throw\ X}}$.
   By the definition of CFG behaviour, we have the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{throw\ x}_b (\bullet_m^{p,x,r}, \sigma)$.
   Then since $k^{p,x}_{\Gamma[m]} = 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet_m^{p,x,r}, \sigma)$ and therefore again: $\theta(c) \xrightarrow{throw\ x} \theta(c')$.

Case $c = (\langle x\rangle_{\mathrm{exc}} \cdot \langle m, p, f, s, z\rangle \cdot A; h)$.

Recall that when there is an exception record on the stack, the JVM takes control, and execution continues with an internal transition. Based on the exception handling table in m we have two sub-cases:

1. Method m has a handler for x ($k^{p,x}_{\Gamma[m]} = t \ \wedge\ t \neq 0$). In this case, applying the abstraction function on the current configuration we obtain: $\theta(c) = (\bullet_m^{p,x}, \sigma)$ where σ again denotes $\gamma(A)$. According to the operational semantics of JVM we have the transition: $c \xrightarrow{catch\ x} c'$ where $c' = (\langle m, t, f, s, z\rangle \cdot A; h)$. From the CFG extraction function for any instruction i that can raise exception x we have: $(\bullet_m^{p,x}, \varepsilon, \circ_m^t) \in \mathcal{G}_{\mathrm{JBC}}^{m,p,i}$.
   Based on the definition of CFG behaviour, this edge induces the transition: $(\bullet_m^{p,x}, \sigma) \xrightarrow{catch\ x}_b (\circ_m^t, \sigma)$.
   Applying the abstraction function on c' we have: $\theta(c') = (\circ_m^t, \sigma)$ and hence: $\theta(c) \xrightarrow{catch\ x} \theta(c')$.
2. There is no handler in m for x ($k^{p,x}_{\Gamma[m]} = 0$). In this case, applying the abstraction function on the current configuration we obtain: $\theta(c) = (\bullet_m^{p,x,r}, \sigma)$. Let $A = \langle n, q, f, s, z\rangle \cdot A'$. Then n is the caller of m, and by the operational semantics of JVM we have: $c \xrightarrow{m\ xret\ n} c'$ where $c' = (\langle x\rangle_{\mathrm{exc}} \cdot \langle n, q, f, s, z\rangle \cdot A'; h')$. From the CFG extraction function for any instruction $i = m[p]$ that can raise exception x we obtain: $(\circ_m^p, \varepsilon, \bullet_m^{p,x,r}) \in \mathcal{G}_{\mathrm{JBC}}^{m,p,i}$.
   Because there is no handler in m, the propagated exception x will be in the CFG extracted for n. So, based on the exception handling table in n we in turn have the following two sub-cases:
   - Method n has a handler for x ($k^{q,x}_{\Gamma[n]} = t \ \wedge\ t \neq 0$). Then according to the CFG extraction rules for n: $(\circ_n^q, m, \bullet_n^{q,x}) \in \mathcal{G}_{\mathrm{JBC}}(n)$.
     By the definition of CFG behaviour: $(\bullet_m^{p,x,r}, \circ_n^q \cdot \sigma') \xrightarrow{m\ xret\ n}_b (\bullet_n^{q,x}, \sigma')$.
     Since $k^{q,x}_{\Gamma[n]} \neq 0$, abstracting c' we obtain: $\theta(c') = (\bullet_n^{q,x}, \sigma')$ and therefore: $\theta(c) \xrightarrow{m\ xret\ n} \theta(c')$.
   - Method n does not define any handler for x ($k^{q,x}_{\Gamma[n]} = 0$). Then according to the CFG extraction rules for n: $(\circ_n^q, m, \bullet_n^{q,x,r}) \in \mathcal{G}_{\mathrm{JBC}}(n)$.
     By the definition of CFG behaviour: $(\bullet_m^{p,x,r}, \circ_n^q \cdot \sigma') \xrightarrow{m\ xret\ n}_b (\bullet_n^{q,x,r}, \sigma')$.
     Since $k^{q,x}_{\Gamma[n]} = 0$, abstracting c' we obtain: $\theta(c') = (\bullet_n^{q,x,r}, \sigma')$ and hence: $\theta(c) \xrightarrow{m\ xret\ n} \theta(c')$. □

4 Extracting control flow graphs from BIR

This section presents the two-phase transformation from Java bytecode to control flow graphs using BIR as intermediate representation. First, we summarize the BIR language, and the transformation from JBC, named `BC2BIR`, as defined by Demange et al. [11]. Next, we describe how BIR is transformed into CFGs. Finally, we prove the soundness for the indirect transformation w.r.t. sequences of methods calls and exceptions.

4.1 The BIR language

The BIR language is an intermediate representation of Java bytecode [11]. The main difference with JBC is that BIR

```
oper    ::= c | null        (constants)
          | f               (field name)
          | C               (class name)
          | pc | pc'        (program counter)
          | m | n | n'      (method name)
expr    ::= c | null
          | expr ⊕ expr
          | tvar | lvar
          | expr.f
lvar    ::= l0 | l1 | ... lj
            this
tvar    ::= t0 | t1 | ... tk
target  ::= lvar
          | tvar
          | expr.f

Assignment ::= target := expr
    Return ::= return expr | return
MethodCall ::= expr.n(expr, ..., expr)
             | target := expr.n(expr, ..., expr)
 NewObject ::= target := new C(expr, ..., expr)
 Assertion ::= notnull expr | notzero expr

  BirInstr ::= nop | if expr pc | goto pc
             | throw expr | mayinit C
             | Assignment | Return
             | MethodCall | NewObject
             | Assertion
```

Fig. 7 Expressions and instructions of BIR

instructions are stack-less, in contrast to bytecode instructions that operate over values stored on the operand stack.

A BIR program is modeled as an environment, in the same way as a JBC program. In fact, the partial mappings for classes and interfaces are common in both representations. They only differ in the definitions of method bodies. The JBC instructions occupy a varying number of positions in the instructions array, while BIR instructions always occupy a single position. Thus, the indexing of corresponding instructions differ, and the transformation has to map the addresses for jumps, branches and exception handlers.

Figure 7 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. The transformation also reconstructs expression trees, i.e., it collapses one-to-many stack-based operations into a single expression. As a result, a JBC program represented in BIR typically has fewer instructions than the original program.

BIR has two types of variables. The first (*lvar*) are identifiers also present in the original bytecode; the second (*tvar*) are new variables introduced by the transformation. Both variables and object fields can be the target of an assignment.

Many of the BIR instructions have an equivalent JBC counterpart, e.g., nop, goto and if. A return *expr* ends the execution of a method with a return value, while return ends a *void* method. 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 BIR's new instruction, which also performs a call to an object constructor.

The throw instruction explicitly transfers control flow to the exception handling mechanism, similarly to the athrow instruction in JBC. Notice that the BIR instruction also has an operand, similarly to the throw X instruction introduced in Sect. 3. Here, a sound static analysis that over-approximates the possible types of the operand (e.g., see [8]) implements the oracle for explicit exceptions.

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

Fig. 8 Implicit exceptions supported by BIR, and associated assertions

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. Thus, the BIR transformation implements the oracle described in the Sect. 3 for the exceptions raised implicitly by the instructions execution. Java bytecode also has class initializers, i.e., the one-time initialization procedures of a class's static fields, invoked when the first object of a class is allocated by a JBC instruction new. If some exception is raised inside the initializer, the JVM captures it, and raises an ExceptionInInitializerError. Thus, BIR adds a special instruction, called mayinit, to indicate that at that point a class initializer may be invoked.

Figure 8 shows all implicit exceptions that are currently supported by the BC2BIR transformation [6], 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 defined below, 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.

Input	Output	Input	Output	Input	Output
`pop`	$\emptyset$	`nop`	`[nop]`	`div`	`[notzero` e_2`]`
`push c`	$\emptyset$	`if` p	`[if e pc']`	`athrow`	`[throw e]`
`dup`	$\emptyset$	`goto` p	`[goto pc']`	`new C`	`[mayinit C]`
`load` l_k	$\emptyset$	`return`	`[return]`	`getfield f`	`[notnull e]`
`add`	$\emptyset$	`vreturn`	`[return e]`		

Input	Output	
`store` l_j	`[`l_j`:=e]` or `[`t_k`:=`l_j`;`l_j`:=e]`	
`putfield f`	`[notnull e;`FSave`(pc,f,`as`);e.f:=e']`	
`invokevirtual n`	`[notnull e;`HSave`(pc,`as`);`t_k`:=e.n(...)]`	
`invokespecial n`	`[`HSave`(pc,`as`);`t_k`:=new C(...)]`	if `n=C` is constructor
	`[notnull e;`HSave`(pc,`as`);`t_k`:=e.n(...)]`	otherwise

Fig. 9 Rules for $\texttt{BC2BIR}_{instr}$

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 their 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 transformation algorithm we refer to [11].

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 algorithm aborts. The function $\texttt{BC2BIR}_{instr}$ is defined as follows.

Definition 6 (*From JBC to BIR*) Let $\mathit{AbsStack} \in \mathit{expr}^*$. The rules defining the *instruction-wise transformation* $\texttt{BC2BIR}_{instr} : \mathbb{N} \times \mathit{JbcInstr} \times \mathit{AbsStack} \to (\mathit{BirInstr}^* \times \mathit{AbsStack}) \cup \{\mathit{Fail}\}$ from JBC into BIR are given in Fig. 9.

As a convention, we use brackets to distinguish BIR instructions from their JBC counterpart. The variables $\texttt{t}_\texttt{k}$ are new and are introduced by the transformation.

JBC instructions `if`, `goto`, `return` and `vreturn` are transformed into corresponding BIR instructions. The `new` is different from `[new C()]` in BIR, it produces only a `[mayinit]` for static initialization of `C`. As mentioned above, object allocation in BIR happens at the same time as initialization, i.e., when the constructor is called. 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 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 whether e is a valid reference; then the auxiliary function *FSave* introduces a set of assignment instructions to temporary variables; followed by the assignment to the field (`e.f`). Similarly, 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. In the first case, there are assignments to temporary variables (*HSave*), followed by the instruction `[new C]` which denotes a call to the constructor. The second case is the same as for `invokevirtual`.

Figure 10 shows the JBC and BIR versions of method `boolean odd(int)` from Fig. 1. The different colors show the collapsing of instructions by the transformation; the underlined instructions are the ones that produce BIR instructions. The BIR method has a local variable (`l0`) and two newly introduced variables (`t0` and `t1`). Notice that the argument for the method invocation and the operand to the `[if]` instructions are reconstructed expression trees. The `[notnull]` instruction asserts that a `NullPointerException` can potentially be raised in the program point 3. The `[mayinit]` instruction shows that class `ArithmeticException` can be initialized at that program point.

4.2 The extraction algorithm from BIR

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

To define $\mathcal{G}_{\text{BIR}}$, we introduce some auxiliary functions, which are similar to the ones introduced for the direct extraction (in Sect. 3). As a convention, we use bars (e.g., $\bar{N}$) to differentiate the corresponding functions from the direct, and indirect algorithms.

The auxiliary function $\bar{k}^{\text{pc},x}_{\Gamma_b[m]}$ returns the first handler (if any) for the exception of type (or a subtype of) x at position

`public static boolean odd(int n)`

	Java bytecode		BIR
0:	`iload_0`		
1:	`ifge 12`	0:	`if ( lx >= 0) goto 5`
4:	`new ArithmeticException`	1:	`mayinit ArithmeticException`
7:	`dup`		
8:	`invokespecial ArithmeticException()`	2:	`t0 := new ArithmeticException()`
		3:	`notnull t0`
11:	`athrow`	4:	`throw t0`
12:	`iload_0`		
13:	`ifne 18`	5:	`if (lx != 0) goto 7`
16:	`iconst_0`		
17:	`ireturn`	6:	`return 0`
18:	`iload_0`		
19:	`iconst_1`		
20:	`isub`	7:	`mayinit EvenOdd`
21:	`invokestatic even(int)`	8:	`t1 := EvenOdd.even(lx - 1)`
24:	`ireturn`	9:	`return t1`

Fig. 10 Comparison between instructions in method `boolean odd(int)`

`pc`. As mentioned before, the function $\bar{\chi}_i$ yields the exception associated with instruction i. The function $\bar{\mathcal{H}}_m^{\text{pc},x,l}$ queries the function $\bar{k}$ for exception handlers in the control point `pc` for the exception type x: if there is any, it returns two edges: one from a normal to an exceptional control node, and one from the exceptional node to the normal node tagged with the handler's initial control point; otherwise, it returns an edge to an exceptional return node. The function also receives a label l as argument, which may be ε, or a method signature.

The extraction is parametrized on a virtual method call resolution algorithm α, in the same fashion as the $\mathcal{G}_{\text{JBC}}$ algorithm presented in Sect. 3.2. The function $res^{\alpha}(\texttt{n'})$ uses α to return a safe over-approximation of the possible receivers to a virtual invocation of a method with signature `n'`, or the single receiver if the signature is from a non-virtual method (e.g., a static method).

Let Γ_b be the environment modeling the BIR representation of a program P, and B_b be the body of some method $\Gamma_b[m]$. Following the definitions for a JBC program in Sect. 3.1, we define for BIR the CFG of a class as the disjoint union of the CFGs of the methods in the class, and the CFG of a program as the disjoint union of all CFGs of the classes in the program, as follows.

Definition 7 (*CFG Extraction from BIR*) The *instruction-wise* extraction function $\mathcal{G}_{\text{BIR}} : (\text{METH} \times \text{ADDR} \times \mathit{BirInstr}) \rightarrow \mathcal{P}(V_m \times L_m \times V_m)$ is defined by the rules in Fig. 11. The method graph for m is defined as $\mathcal{G}_{\text{BIR}}(m) = \bigcup_{(p,i)\in B_b} \mathcal{G}_{\text{BIR}}^{m,p,i}$. The control flow graph for the program is defined as $\mathcal{G}_{\text{BIR}}(\Gamma_b) = \bigcup_{\{m|\Gamma_b[m]\in\Gamma_b\}} \mathcal{G}_{\text{BIR}}(m)$.

We subdivide the definition of $\mathcal{G}_{\text{BIR}}$ 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 $\bar{\mathcal{N}}_m^{\text{pc},n}$, which returns 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.

First, we describe the rules applied by 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 BIR transformation provides the static type *staticT* $(expr)$ of the exception raised by `[throw` *expr*`]`, and we soundly over-approximate the possible types of *expr* to all its subtypes. The extraction produces an exceptional edge for each type, 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 Fig. 8), 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 `NullPointer Exception` [27, § 6.5].

$$
\bar{\mathcal{H}}_m^{\texttt{pc},x,l} = \begin{cases} \{\ (\circ_m^{\texttt{pc}}, l, \bullet_m^{\texttt{pc},x}), (\bullet_m^{\texttt{pc},x}, \varepsilon, \circ_m^{\texttt{pc'}})\ \} & \text{if } \bar{k}_{\Gamma_b[m]}^{\texttt{pc},x} = \texttt{pc'} \neq 0 \\ \{\ (\circ_m^{\texttt{pc},x}, l, \bullet_m^{\texttt{pc},x,r})\ \} & \text{if } \bar{k}_{\Gamma_b[m]}^{\texttt{pc},x} = 0 \end{cases}
$$

$$
\bar{\mathcal{N}}_m^{\texttt{pc},n} = \bigcup_{\{x \mid \bullet_n^{\texttt{pc'},x,r} \in V_n\}} \bar{\mathcal{H}}_m^{\texttt{pc},x,n}
$$

$$
\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},i} = \begin{cases} \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc+1}})\} & \text{if } i \in \mathit{Assignment} \cup \{\texttt{[nop]}\} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc+1}}), (\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc'}})\} & \text{if } i = \texttt{[if } \mathit{expr}\ \texttt{pc']} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc'}})\} & \text{if } i = \texttt{[goto pc']} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc},r})\} & \text{if } i \in \mathit{Return} \\ \bigcup_{\{x \mid x <:_{static} T(expr)\}} \bar{\mathcal{H}}_m^{\texttt{pc},x,\varepsilon} & \text{if } i = \texttt{[throw } \mathit{expr}\texttt{]} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc+1}})\} \cup \bar{\mathcal{H}}_m^{\texttt{pc},\bar{\chi}_i,\varepsilon} & \text{if } i \in \mathit{Assertion} \\ \{(\circ_m^{\texttt{pc}}, \texttt{C}, \circ_m^{\texttt{pc+1}})\} \cup \bar{\mathcal{H}}_m^{\texttt{pc},\texttt{N.P.E.},\varepsilon} \cup \bar{\mathcal{N}}_m^{\texttt{pc},\texttt{C}} & \text{if } i \in \mathit{NewObject} \\ \bigcup_{n \in res^{\alpha}(\texttt{n'})} \{(\circ_m^{\texttt{pc}}, n, \circ_m^{\texttt{pc+1}})\} \cup \bar{\mathcal{N}}_m^{\texttt{pc},n} & \text{if } i \in \mathit{MethodCall} \end{cases}
$$

Fig. 11 Extraction rules for control flow graphs from BIR

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

Next, we describe the inter-procedural analysis. In all program points where there is a method invocation, the function $\bar{\mathcal{N}}_m^{\texttt{pc},n}$ adds exceptional edges, relative to exceptions propagated by called methods. It analyses if the CFG of an invoked method n contains an exceptional return node. If it does, then function $\bar{\mathcal{H}}_m^{\texttt{pc},x,n}$ verifies whether the exception of type x is caught in position pc. If so, it adds two edges: one labelled with the signature of the called method n, showing that it has terminated with an uncaught exception, and a second edge showing the transfer of control to the exception handler. Otherwise it adds an edge to an exceptional return node. In the latter case, the propagation of the exception continues until it is caught by some caller method, or there are no more methods to handle it. This is similar to the process described by Jo and Chang [23], who also present a fixed-point algorithm to compute the propagation edges. It checks the pre-computed call-graph to determine at which control points the invocations are made to a method propagating a given exception. If there is a suitable handler for that exception, it adds the respective handling edges, and the process stops. Otherwise, the computation proceeds.

4.3 Soundness of CFG extraction

This section discusses the correctness proof of the CFG extraction algorithm. We start by pointing out that BIR has its operational semantics defined [11]. Moreover, it was proven that there is a semantic-preserving relation between the terminating traces for the same program in JBC and BIR. However, our algorithm is defined purely syntactically, thus we do not use BIR's correctness result.

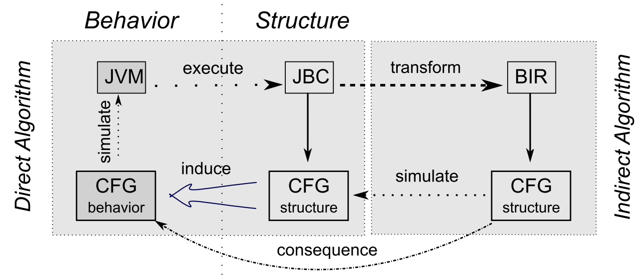

Fig. 12 Schema for CFG extraction and correctness proof

We prove correctness indirectly, using the idealized direct extraction algorithm $\mathcal{G}_{\text{JBC}}$ defined in Sect. 3 as a reference. $\mathcal{G}_{\text{JBC}}$ 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 this idealized algorithm by proving that given a JBC program, the CFG produced by our extraction algorithm ($\mathcal{G}_{\text{BIR}} \circ$ BC2BIR) structurally simulates the CFG produced by the direct extraction algorithm ($\mathcal{G}_{\text{JBC}}$). We then reuse a result established previously by Huisman et al. [20, Theorem 1] that structural simulation entails behavioural simulation. As explained in Sect. 2, the latter result is stated over slightly different, but equivalent, definitions of CFG and CFG behaviour, and thus the result applies in our setting as well. By transitivity of simulation we conclude that the behaviour induced by the CFG extracted by $\mathcal{G}_{\text{BIR}} \circ$ BC2BIR simulates the JVM behaviour. Figure 12 summarizes our approach.

We sketch here the overall proof of structural simulation, and discuss two cases (for the ThrInst and RetInst groups) in full detail. The remaining cases are given in Appendix 11. Before discussing the proof sketch, we first introduce some terminology and make some relevant observations.

Preliminaries for the Correctness Proof The BC2BIR transformation may collapse several bytecode instructions into a

single BIR instruction. We divide the JBC instructions into two sets: the *producer* instructions, i.e., those that produce at least one BIR instruction in function $\texttt{BC2BIR}_{instr}$, and the *auxiliary* ones, i.e., those that produce none. This division can be deduced from Fig. 9 (on page 22). 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 sub sequences delimited by producer instructions. Thus each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Equivalently, we may say that a segment is defined by an optional *basic block* [1] and a subsequent producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java Bytecode Verifier (see Sect. 2.1). All methods in such programs must have `goto`, `return`, or `athrow` as the last (reachable) instruction, which are producer instructions. Therefore, there can not be contiguous (reachable) instructions that are not delimited by a producer instruction.

Each bytecode segment is transformed into a set of contiguous instructions by `BC2BIR`. We call this set a *BIR segment*, which is a partition of the BIR instruction array. There exists a one-to-one mapping between bytecode segments and the BIR segments, which is also order-preserving. Thus, we can associate each instruction, either in the JBC or BIR arrays, to the unique index of its correspondent bytecode segment. Figure 10 (on page 23) illustrates the partitioning of instructions into segments. Method `odd` has eight bytecode (and BIR) segments, as indicated by the distinct shades. Producer instructions are underlined.

In the definition of the direct extraction algorithm in Fig. 6 (on page 13), 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-like graph of internal transfer edges of the same length as the number of auxiliary instructions, followed by the edges generated for the producer instruction. Let p be the position for the first auxiliary instruction, and q the position of the producer instruction. We illustrate the pattern for this path-like graph below.

$$\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)} \xrightarrow{\varepsilon} \circ_m^{succ(succ(p))} \xrightarrow{\varepsilon} \cdots \circ_m^q$$

It is easy to see that the path-like graph is weakly simulated by some reflexive edge $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc}}$. Therefore, for simplicity we present the proof for the case where $p = q$. That is, for JBC segments without auxiliary instructions.

Another important observation is about the mapping between control addresses between the JBC and the BIR representations. A control address q from a branching instruction (e.g., `goto q` or `ifeq q`) or from an exception handler is always mapped to the first control address (let's call it `pc`) of the corresponding BIR segment that q belongs to. This is necessarily the case because either q contains a producer instruction, which will generate a set of sequential BIR instructions, with smallest control address being `pc`, or it contains an auxiliary instruction, which will be collapsed into the BIR instructions when the first JBC producer instruction is processed, also having `pc` as the control address with smallest index.

Based on the observations above, our main theorem states that the method graphs extracted using the indirect algorithm weakly simulate (see Definition 8 in Appendix 10) the method graphs using the direct algorithm. The abstract stacks are omitted in the proof since examining the instructions is sufficient to produce the edges.

Theorem 2 (Structural Simulation of CFGs) *Let Γ be the environment modeling a well-formed JBC program P_{jbc}. Then $(\mathcal{G}_{\mathrm{BIR}} \circ \texttt{BC2BIR})(\Gamma)$ weakly simulates $\mathcal{G}_{\mathrm{JBC}}(\Gamma)$.*

Proof (Sketch) Let m be a method signature and $\Gamma[m]$ be the method's definition in Γ. Let p range over indices in the bytecode instructions array, `pc` over indices in the BIR instructions array, $\circ_m^{p,x,r}$ over control nodes in $\mathcal{G}_{\mathrm{JBC}}(\Gamma[m])$, and $\circ_m^{\texttt{pc},x,r}$ over control nodes in $(\mathcal{G}_{\mathrm{BIR}} \circ \texttt{BC2BIR})(\Gamma[m])$. The control nodes are valuated with two optional atomic propositions: x, which is an exception type, and r, which is the atomic proposition denoting a return point. Further, let $seg_{JBC}(m, p)$ and $seg_{BIR}(m, \texttt{pc})$ be two auxiliary functions that return the segment index that a JBC, or a BIR control address belongs to, respectively. Let s be the index of a BIR segment. Function $fst(s)$ return its first control address and $oap(s, XR)$ return the set of control addresses in s tagging a node with the non-empty set of *optional atomic propositions* equal to XR.

We define the binary relation R as the union of two binary relations R_1 and R_2, as follows:

$$\begin{aligned}
R &\stackrel{def}{=} R_1 \cup R_2\\
R_1 &\stackrel{def}{=} \{(\circ_m^p, \circ_m^{\texttt{pc}}) \mid seg_{JBC}(m, p) = seg_{BIR}(m, \texttt{pc})\\
&\qquad \wedge \texttt{pc} = fst(seg_{BIR}(m, \texttt{pc}))\}\\
R_2 &\stackrel{def}{=} \{(\circ_m^{p,XR}, \circ_m^{\texttt{pc},XR}) \mid seg_{JBC}(m, p) = seg_{BIR}(m, \texttt{pc})\\
&\qquad \wedge \texttt{pc} \in oap(seg_{BIR}(m, \texttt{pc}), XR)\}
\end{aligned}$$

and show the relation to be a weak simulation in the standard fashion, following Proposition 1 (to be found in Appendix 10): for every pair of nodes in R, we first show that the nodes have the same set of atomic propositions, and then match every strong edge that has the first node as source, to a corresponding weak edge that has the second node as source, so that the target nodes are again related by R.

Intuitively, the direct and indirect algorithms extract a similar branching structure for the same JBC code segment,

differing in the occurrences of silent transitions. Therefore, R relates the first normal (source) nodes extracted by both algorithms (i.e., R_1), and a node from the direct algorithm tagged with a non-empty set of atomic propositions to nodes extracted in the indirect algorithm with the same set of atomic propositions (i.e., R_2). Notice that the only case where the indirect algorithm produces two distinct nodes with the same set of non-empty atomic propositions is for the case of method invocations, where a `N.P.E.` may be raised in different control addresses: either by a `[notnull]` or propagated by the callee method.

Let $(\circ^p_m, \star) \in R_1$ and let $(\circ^{p,r}_m, \star) \in R_2$ where $\circ^p_m$ and $\circ^{p,r}_m$ are control nodes in $\mathcal{G}_{\text{JBC}}(\Gamma[m])$ and $\star$ is a control node in $(\mathcal{G}_{\text{BIR}} \circ \texttt{BC2BIR})(\Gamma[m])$. We consider the two cases separately. Let first $(\circ^p_m, \star) \in R_1$. The proof proceeds by *case analysis* on the *type of the producer instruction* of the bytecode segment $seg_{JBC}(m, p)$ giving rise to $\circ^p_m$. The cases follow the subsets of JBC instructions presented in Fig. 4, which share the same extraction rule in the direct algorithm for its instructions. We rely on the notation to indicate that two nodes have the same set of atomic propositions.

We now present the case for the ThrInst. Let X be the set of all possible exception types for an instance of the idealized `throw X`, which is the single instruction in ThrInst. Then $\texttt{X} \subseteq \{x | x <: staticT(\texttt{e})\}$ for the corresponding instance of `[throw e]`. That is, the set of possible exceptions in the indirect extraction soundly over-approximates the set X since any exception $x \in \texttt{X}$ is necessarily a sub-type of $staticT(\texttt{e})$. Let $x \in \texttt{X}$.

The direct extraction for the `throw` instruction produces two edges if there is a suitable handler for the exception x in position p. Otherwise, it produces a single edge, whose sink node is an exceptional return node:

$$\mathcal{G}^{m,p,\texttt{throw X}}_{\text{JBC}} = \begin{cases} \{\circ^p_m \xrightarrow{\varepsilon} \bullet^{p,x}_m, \bullet^{p,x}_m \xrightarrow{\varepsilon} \circ^q_m\} & \text{if has handler} \\ \{\circ^p_m \xrightarrow{\varepsilon} \bullet^{p,x,r}_m\} & \text{otherwise} \end{cases}$$

The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction. Then, similarly to $\mathcal{G}_{\text{JBC}}$, the $\mathcal{G}_{\text{BIR}}$ function produces either one, or two edges:

$$\texttt{BC2BIR}^{p,\texttt{athrow}}_{instr} = [\texttt{throw e}]$$

$$\mathcal{G}^{m,\texttt{pc},[\texttt{throw e}]}_{\text{BIR}} = \begin{cases} \{\circ^{\texttt{pc}}_m \xrightarrow{\varepsilon} \bullet^{\texttt{pc},x}_m, \bullet^{\texttt{pc},x}_m \xrightarrow{\varepsilon} \circ^{\texttt{pc}'}_m\} & \text{if has handler} \\ \{\circ^{\texttt{pc}}_m \xrightarrow{\varepsilon} \bullet^{\texttt{pc},x,r}_m\} & \text{otherwise} \end{cases}$$

Then $\star = \circ^{\texttt{pc}}_m$ since $\texttt{pc} = fst(s)$. In the case where there is an exception handler for x in p and pc, the edge $\circ^p_m \xrightarrow{\varepsilon} \bullet^{p,x}_m$ is matched by the corresponding weak edge $\circ^{\texttt{pc}}_m \Longrightarrow \bullet^{\texttt{pc},x}_m$. Then also $(\bullet^{p,x}_m, \bullet^{\texttt{pc},x}_m) \in R$ since $seg_{JBC}(p) = seg_{BIR}(m, \texttt{pc})$ and $\texttt{pc} \in oap(s, \{x\})$. That is, pc tags a node where the set $XP = \{x\}$. Actually, for this case there is only one node since pc is the only control address in the segment. Further, there is the edge $\bullet^{p,x}_m \xrightarrow{\varepsilon} \circ^q_m$, which is matched by $\bullet^{\texttt{pc},x}_m \Longrightarrow \circ^{\texttt{pc}'}_m$, and again $(\circ^q_m, \circ^{\texttt{pc}'}_m) \in R$ since $seg_{JBC}(m, q) = seg_{BIR}(m, \texttt{pc}')$ and $\texttt{pc}' = fst(seg_{BIR}(m, \texttt{pc}'))$. That is, pc′ is the first control address on its code segment.

In the case where there is no exception handler for x, the only edge is $\circ^p_m \xrightarrow{\varepsilon} \bullet^{p,x,r}_m$, which is matched by $\circ^{\texttt{pc}}_m \Longrightarrow \bullet^{\texttt{pc},x,r}_m$. Moreover, $(\bullet^{p,x,r}_m, \bullet^{\texttt{pc},x,r}_m) \in R$ since $seg_{JBC}(p) = seg_{BIR}(m, \texttt{pc})$ and $\texttt{pc} \in oap(s, \{x, r\})$. That is, pc tags the node where the set $XP = \{x, r\}$, which concludes the case.

Let now $(\circ^{p,r}_m, \star) \in R_2$. The proof proceeds with the RetInst set, the only type of the producer instructions of the bytecode segment $seg_{JBC}(m, p)$ giving rise to $\circ^{p,r}_m$. All return instructions are producer instructions. However, the direct algorithm does not produce edges for them, but simply adds the atomic proposition r to the normal sink nodes tagged with the address of the return instruction. Let p be the address of the return instruction. The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction, applied to which $\mathcal{G}_{\text{BIR}}$ produces a single edge:

$$\texttt{BC2BIR}^{p,\texttt{return}}_{instr} = [\texttt{return } expr]$$

$$\mathcal{G}^{m,\texttt{pc},[\texttt{return } expr]}_{\text{BIR}} = \{\circ^{\texttt{pc}}_m \xrightarrow{\varepsilon} \circ^{\texttt{pc},r}_m\}$$

In this case we have to relate $\circ^{p,r}_m$ via R_2 rather than via R_1. Then $\star = \circ^{\texttt{pc},r}_m$ since $seg_{JBC}(p) = seg_{BIR}(m, \texttt{pc})$, and $\texttt{pc} \in oap(s, \{r\})$. That is, pc tags the only node where the set $XP = \{r\}$. Since there is no outgoing edge from $\circ^{p,r}_m$, this concludes the case of return instructions and the whole proof. □

5 The ConFlEx tool

The indirect extraction algorithm is implemented as our CFG extraction tool ConFlEx [15]. It uses Sawja [19], a library for the static analysis of Java bytecode programs. Sawja features the most popular virtual method call resolution algorithms, and the transformation from bytecode into BIR. However, the standard implementation performs only a syntactic transformation, and does not compute the object types. We have instrumented the BIR transformation from Sawja to compute the most generic object type. Thus, we can estimate soundly the type of explicit exceptions.

ConFlEx implements the extraction rules from Fig. 11 (on page 24), including the computation of exception propagation. Currently the extraction is fine-tuned for the particular variant of control flow graphs, presented in Definition 1. However, it can be extended to support other formalizations with a relatively small effort. Also, the control node addresses are relative to the BIR representation of the analysed pro-

Table 1 Statistics for CONFLEX

Program	# 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	30,930	10,850	524	20,595	20,937	262	27267	27,717	35
Java-Cup	31,958	13,174	679	24,263	24,549	262	32031	32,449	43
TJWS	33,653	13,129	423	32,333	34,325	515	71218	78,233	354
JFlex	53,474	20,433	1191	39,429	40,035	675	53956	54,777	102
JXplorer	186,644	77,536	1129	150,495	154,583	21,450	332366	344,095	2036
Xowa	713,160	294,339	77,468	716,362	719,534	146,019	5741597	5,865,206	396,437
Soot	1,345,574	516,404	64,856	1,081,297	1,081,962	4,055,987	14,307,406	14,319,126	3,665,739

gram. Though possible, it requires a medium effort to implement the mapping from the extracted CFG to the original JBC addresses.

We have evaluated the performance of CONFLEX. It is future work to also evaluate the precision of the extracted CFGs in terms of the ratio of spurious nodes, and to experiment with different VMC resolution algorithms. To evaluate our tool, we have applied it to a collection of real-world applications. The main criterion when selecting these applications has been their size (number of instructions), without verification in mind. We aimed to diversify the type of applications. For instance, JFlex and Java-Cup are parsing tools, TJWS is a web server, and JExplorer is an LDAP browser. In our evaluation we use the Rapid Type Analysis [5], which provides the best balance between performance and precision [38]. 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. Table 1 shows statistics about the analysed programs, the size of the produced CFGs and the extraction times.

BIR time stands for the translation of the JBC program into the BIR representation, which is executed after the virtual method call resolution. The translation time is close to linear in the number of JBC instructions, in all cases except for the code of the Xowa web browser. Also, we can observe that the number of BIR instructions is less than 42% of that of bytecode instructions, for all cases. The average number of nodes produced in the *intra-procedural* computation is roughly twice the number of BIR instructions. That is, about half of the control nodes are exceptional nodes, which is a consequence of the over-approximation of the exceptional flow.

We can also observe 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 analysed program. Surprisingly, the inter-procedural analysis, which is a fixed-point computation, showed in practice to only take a fraction of the computation time of the intra-procedural analysis, with Xowa again being the only exception. We conjecture that the deviation for Xowa is caused by a more significant presence of exceptional flows in its implementation.

We do not provide comparative data with other extraction tools such as Soot [39] or Wala [40], since this would demand the implementation of extraction rules corresponding to ours, but from their intermediate representations. However, experimental results from SAWJA [19] 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.

6 Discussion

In this section we describe some implications of our algorithm for control flow graph extraction. First, we address its precision, and the supported subset of implicit exceptions. Next, we describe a natural extension of the algorithm for incremental CFG extraction. Finally, we speculate on the extraction of control flow graphs from multi-threaded programs.

6.1 Precision of the extraction

The present work focuses on the soundness of the CFG extraction algorithm. As usual for static analyses, soundness (often also referred to as safety) comes at the expense of precision. Still, we conjecture that the CFGs extracted by our algorithm are the most precise ones that can be obtained when abstracting from all non-exceptional data and relativizing on an (externally provided) sound virtual method call resolution algorithm. While a formal proof of this conjecture

is beyond the scope of this paper, we provide below some intuitive justification.

In our analysis, exceptions are the only data type considered. However, the potential occurrence of implicit (runtime) exceptions depends heavily on program data. For instance, whether an `ArithmeticException` is raised because of a division by zero, or whether an `ArrayIndexOutOfBoundsException` is raised because of an array access with an invalid index, depends on the instruction's operands. Our extraction algorithm does not consider such data, therefore it necessarily over-approximates the implicit exceptions of every Java bytecode instruction to the ones it may potentially raise, as defined in the JVM specification [27]. The necessary information for this is obtained from the assertions (see Fig. 8) introduced by the BIR transformation.

However, the processing of explicit exceptions in our extraction algorithm uses type analysis to infer the possible exception types for a given `athrow` instruction. The symbolic execution performed by the BIR transformation analyses the operand stack, and associates a variable to the raised exception. We over-approximate the exception types of a given `[throw e]` to its static type and all subclasses of the static type.

As explained above, both extraction algorithms are parametrized by a sound virtual method call resolution algorithm. Thus, the indirect extraction algorithm inherits the imprecision of the virtual method call resolution algorithm. This may impact negatively the size of the extracted CFGs by adding superfluous call edges; moreover, these call edges may give rise to additional exceptional nodes and edges, caused by the propagation of exceptions by the callee sites.

The primary utility of our CFGs is the verification of control-flow-based temporal safety properties. This allows the CFG extraction to abstract from most of the data. This design choice gives rise to a comparatively lightweight approach that is both efficient and easy to be shown sound. Still, our extraction algorithm can benefit from finer data flow analyses such as *null pointer* analysis [37] or symbolic execution [25], provided that these analyses are also proven to be sound. Furthermore, the latter technique is envisaged in [34], by one of the present authors, as the means to extract CFGs with symbolic data.

6.2 Exceptions supported by the extraction

As explained in Sect. 4, the concrete extraction algorithm $\mathcal{G}_{\text{BIR}} \circ$ `BC2BIR` extracts CFGs soundly, considering all explicit exceptions, i.e., raised by `athrow`, and a subset of the implicit exceptions (Fig. 8). Here we clarify which implicit exceptions are supported, and what are the implications of supporting the full set of exceptions.

We define as *implicit exceptions* all those exceptions raised by the Java Virtual Machine, either asynchronously, by signaling an internal error in the JVM implementation (`VirtualMachineError`); or synchronously, because of a linking error (`LinkageError`), the exhaustion of some resource (e.g., `OutOfMemoryError`), or an abnormal instruction execution, e.g., a division by zero (`ArithmeticException`).

Many implicit exceptions are raised during the execution of a particular instruction. However, there are also implicit exceptions that depend on the JVM execution settings (e.g., allocated memory), its implementation, or the hosting machine. All these exceptions are subtypes of the class `java.lang.Error`, which according to the Java specification [27] represent severe problems, and applications should not try to recover from them. The goal of our tool is to extract models for typical software engineering processes, such as formal software verification. Adding exceptions relative to the execution environment, which can be raised virtually at any control point, would result in bloated CFGs that do not contribute to the goal of verification. Therefore, our algorithm does not support implicit exceptions that are subtypes of `Error`. The single outlier is `ExceptionInInitializerError` since it is not raised by an environment anomaly, but due to an exception raised by the execution of a class initializer.

On the other hand, the implicit exceptions caused by abnormal execution of an instruction are supported. Demange et al. [11] have made a careful analysis of the official Java specification, and encode exceptions by means of BIR assertions. Currently, `IllegalMonitorStateException` is the only unsupported exception. It is intrinsic to concurrent programs, and does not impact the algorithms described in this paper, which are defined for sequential programs. Nevertheless, our algorithm can be easily adapted to accommodate this, or any potentially new exception that newer versions of BIR may support by means of assertions.

We conclude by mentioning that our tool, ConFlex, supports the full set of exceptions, as defined in the standard Java API. However, the user must explicitly indicate that API methods are part of the analysed program since the parsing and extraction of the such methods impact the analysis time and the CFGs' size. Therefore, the default setting is to skip API methods. In that case, if an API method is invoked inside a program, the tool still considers the exceptions listed in the `throws` clause of the method declaration (as illustrated by Fig. 3, where the `parseInt` method declares that it may propagate a `NumberFormatException`).

6.3 Incremental extraction

The program model considered here is in essence procedure-modular: control flow graphs of programs are simply collec-

tions of method graphs. These method graphs are obtained by a combination of an intra- and an inter-procedural analysis, where the former is based solely on the code of the method at hand. Thus, a natural question that arises is whether and how the present framework can be extended to accommodate the *incremental* extraction of CFGs, starting from an incomplete JBC program and refining the extracted models as more code becomes available. This question was addressed by two of the present authors in Gomes et al. [16]. We briefly summarize the approach described in this paper.

Gomes et al. consider programs where the components are statically defined, but the implementation of some components is not yet available. They develop an algorithm $\circ\mathcal{G}$ for incomplete programs as a generalization of the present $\mathcal{G}_{\text{BIR}}$ algorithm. The inter-dependencies involving unavailable components are captured by means of *user-provided interfaces* that specify the methods that can be called and the exceptions that will be handled by the component. The VMC resolution algorithm is fixed to *Modular Class Analysis* (MCA), a variant of the *Class Hierarchy Analysis* (CHA) [10] for incomplete programs with annotations. Figure 13 shows the extraction rules for $\circ\mathcal{G}$, where the shaded parts highlight the extension of $\mathcal{G}_{\text{BIR}}$ for unavailable code.

The modular framework defines formally the constraints on instantiating yet unavailable code, needed to ensure the soundness of the already generated CFGs from the available components. The soundness of the framework is established by showing that the CFGs extracted by $\circ\mathcal{G}$ from the available methods of an incomplete program simulate the CFGs for the same methods in a complete program extracted by $\mathcal{G}_{\text{BIR}}$, if the instantiation of missing code respects the defined constrains. This result is then combined with the theoretical results presented in this paper to conclude that the CFGs extracted with the $\circ\mathcal{G}$ algorithm are also sound w.r.t. the JBC behaviour.

ConFlEx supports the incremental extraction and refinement of CFGs. It features caching of previous analyses, and matching of newly arriving code against their interface specifications. Experimental results confirm the intuitive expectation that the over-approximation resulting from analysing incomplete programs can have a significant negative impact on the size of the extracted CFGs.

Even though being modular, the described technique is restricted to programs for which we know in advance all its components. For certain types of systems, such as *open* systems, such an assumption cannot be made. We therefore plan in future work to investigate how to generate (rather than specify) the constraints on the unavailable parts of the program, needed to ensure a given global property.

6.4 Multi-threaded control flow

The definitions and algorithm presented above concern only sequential control flow. However, ConFlEx can also be used to analyse concurrent programs, by extracting CFGs from individual threads: the user simply has to provide the thread's entry method as an argument, which invariably is the `run()` method from a subtype of `Runnable`. However, at present the produced CFGs do not take into account the interaction between the threads. Thus, a natural extension of our framework and tool would be to support a richer program model that captures the relevant aspects of concurrency.

$$
MCA(\texttt{C.ns}) = \begin{cases} \{\, c'.\texttt{ns} \mid c' \text{ is the closest super-type s.t. } c'.\texttt{ns} \in dom(\Gamma^M)\} & \text{if call is virtual} \\ \cup \{\, c.\texttt{ns} \mid c <: \texttt{C} \wedge c.\texttt{ns} \in dom(\Gamma^M)\,\} & \\ \{\, \texttt{C.ns} \,\} & \text{otherwise} \end{cases}
$$

$$
\mathcal{H}_m^{\texttt{pc},x,l} = \begin{cases} \{\, (\circ_m^{\texttt{pc}}, l, \bullet_m^{\texttt{pc},x}), (\bullet_m^{\texttt{pc},x}, \varepsilon, \circ_m^{\texttt{pc'}})\,\} & \text{if } k_{\Gamma[m]}^{\texttt{pc},x} = \texttt{pc'} \neq 0 \\ \{\, (\circ_m^{\texttt{pc},x}, l, \bullet_m^{\texttt{pc},x,r})\,\} & \text{if } k_{\Gamma[m]}^{\texttt{pc},x} = 0 \end{cases}
$$

$$
\mathcal{N}_m^{\texttt{pc},n} = \begin{cases} \bigcup_{\{x \mid \bullet_n^{\texttt{pc'},x,r} \in V_n\}} \mathcal{H}_m^{\texttt{pc},x,n} & \text{if } \Gamma^M[m] \text{ is available} \\ \bigcup_{x \in E_{\Gamma_o} - \Gamma_o^m[n].\texttt{handlers}} \mathcal{H}_m^{\texttt{pc},x,n} & \text{otherwise} \end{cases}
$$

$$
\circ\mathcal{G}_m^{\texttt{pc},i} = \begin{cases} \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc+1}})\} & \text{if } i \in \mathit{Assignment} \cup \{\texttt{[nop]}\} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc+1}}), (\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc'}})\} & \text{if } i = \texttt{[if } \mathit{expr} \texttt{ pc']} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc'}})\} & \text{if } i = \texttt{[goto pc']} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc},r})\} & \text{if } i \in \mathit{Return} \\ \bigcup_{\{x \mid x <: \mathit{staticT}(\mathit{expr})\}} \mathcal{H}_m^{\texttt{pc},x,\varepsilon} & \text{if } i = \texttt{[throw } \mathit{expr}\texttt{]} \\ \{(\circ_m^{\texttt{pc}}, \varepsilon, \circ_m^{\texttt{pc+1}})\} \cup \mathcal{H}_m^{\texttt{pc},\bar{\chi}_i,\varepsilon} & \text{if } i \in \mathit{Assertion} \\ \{(\circ_m^{\texttt{pc}}, \texttt{C}, \circ_m^{\texttt{pc+1}})\} \cup \mathcal{H}_m^{\texttt{pc},\texttt{N.P.E.},\varepsilon} \cup \mathcal{N}_m^{\texttt{pc},\texttt{C}} & \text{if } i \in \mathit{NewObject} \\ \bigcup_{n \in MCA(\texttt{n'})} \{(\circ_m^{\texttt{pc}}, n, \circ_m^{\texttt{pc+1}})\} \cup \mathcal{N}_m^{\texttt{pc},n} & \text{if } i \in \mathit{MethodCall} \end{cases}
$$

Fig. 13 CFG extraction rules from incomplete BIR programs

In previous work by two of the present authors, Huisman et al. define a notion of CFGs for multithreaded programs [20, Definition 9]. It considers the following primitives for thread synchronization in JBC programs: thread spawning and joining, lock acquiring and releasing, and wait, notify, and notifyAll.

The definition presents certain difficulties for the sound CFG extraction from Java bytecode. The major one is that it requires the sets of threads and of locks to be finite. However, Java allows unbounded creation of objects, thus an unbounded number of threads and locks. Hence, a strategy is needed to soundly over-approximate the sets of threads and locks used in the program.

A first approach to over-approximate the set of threads would be to abstract all threads of the same type (i.e., that are instances of the same class) into a single representative. Though this is a safe over-approximation, it may be too imprecise for verification purposes. For instance, a spurious violation of mutual exclusion is reported when two (or more) threads of the same type are abstracted into the same representative, and the abstract thread enters a critical section.

The situation with locks is similar. A simple example with two threads (possibly not from the same type) sharing two locks shows that a deadlock would not be detected. Let us assume that each tread must hold both locks to enter the critical section. If locks are over-approximated into a single representative, a thread that acquired one lock is abstracted as having both. This hides a potential deadlock, where each thread holds only one of the locks, but blocks while waiting for the other lock to be released.

One can refine the above idea by abstracting into a user-provided upper bound of representative threads and locks per type. This would rule out the most common spurious errors, and provide a higher precision as the bound increases. However, further investigations are needed to define strategies for providing such an upper bound for common practical cases.

7 Related work

Java bytecode has several object-oriented aspects 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 programs. To the best of our knowledge, for none of the existing extraction algorithms a correctness proof has been provided.

Zhao [41] presents an initial formal definition of CFGs for Java bytecode programs. However, there is no formal definition of an extraction algorithm, only a description of the relevant aspects of the transformation from JBC to the control flow graphs. Among these, the paper considers exceptions and describes how exceptions are handled. However, it does not discuss how they are raised, or how to estimate the exception type.

Sinha et al. [32,33] 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. [22] extend the work of Sinha et al. to C++ source code. C++ has the same scheme of `try-catch` and exception propagation as Java source, but without the `finally` blocks, or implicit exceptions. This work does not consider the exception 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. Instead, our work considers the exception types, and thus produces finer CFGs. Moreover, it tells which exceptions can be raised, or propagated from method invocations.

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

Jo and Chang [23] 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.

Recently, Mihancea and Minea [28] presented JMODEX, a tool for the extraction of finite-state models from Java web applications that are tailored for the model checking of security properties. In contrast to CONFLEX, their tool does not fully support virtual method calls, nor exceptional flow.

Finally, we cite Bandera [9,12] as a pioneering tool to generate abstract models from Java source programs for model checking. It contains several features, such as output for multiple model checkers, and some static analyses, such as *slicing*. In comparison to ConFlEx, Bandera is a versatile tool, which provides an integrated framework to program checking. The work mentions the support of exceptions as future work. However, we could not find other references about exceptions in further publications about Bandera.

8 Conclusion

This paper presents an efficient and precise control flow graph extraction algorithm for sequential Java bytecode programs that considers normal as well as exceptional control flow. The main contribution of the paper is a formalization and a soundness proof, providing a formal argument for why the algorithm is correct, in the sense that it extracts control flow graphs whose behaviour (in terms of sequential program executions) soundly over-approximates the behaviour 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. Additionally, the paper also contributes an implementation of the algorithm, whose utility and efficiency has been demonstrated on several test cases.

The algorithm is developed in two stages, first abstracting away from the complications presented by exceptions in stack-based languages by means of an (exceptions-predicting) oracle. The resulting idealized algorithm serves as a specification for concrete, efficient instantiations, which have to realize the oracle suitably. In the second stage, we present a concrete algorithm that not only implements the oracle, but also gives rise to more compact CFGs. Both goals are achieved by means of a translation to the intermediate, stack-less Java bytecode representation BIR, supported by the Sawja tool. The CFGs are then extracted from the BIR representation through a combination of intra-procedural and inter-procedural analyses. The constructed CFGs are precise, relative to the chosen algorithm for virtual method call resolution and to static analyses that abstract from all data but exceptions.

The correctness of the idealized extraction algorithm is proved directly, by means of a simulation relation between the behaviour of the original program and the behaviour of the extracted CFG. Proving correctness of the concrete algorithm is then simplified considerably: one has only to prove structural (rather than behavioural) simulation between the CFGs extracted by the idealized algorithm and the CFGs extracted by the concrete one. Still, the proof is not trivial, and requires the introduction of the notion of bytecode segment, in order to match bytecode instructions to BIR instructions. Structural simulation is then shown for the CFGs extracted by the two algorithms for each segment of the original program, by case analysis on the type of the last instruction in the segment.

The concrete CFG extraction algorithm has been implemented as the ConFlEx tool. The experimental results show that the algorithm is reasonably efficient, and that it produces compact CFGs. However, both performance and precision are highly dependent on the program implementation, and specifically on its exceptional flow, and the size of the call graph.

9 Appendix: Correctness of $\mathcal{G}_{\text{JBC}}$

Now we enunciate the Theorem 1 again, followed by its full correctness proof.

Theorem (CFG Soundness) *Let P_{jbc} be a well-formed Java bytecode program modeled by the environment Γ. The behaviour of the extracted flow graph $\mathcal{G}_{\text{JBC}}(\Gamma)$ simulates the execution of P_{jbc}.*

Proof We prove *simulation* between the execution of P_{jbc} and the behaviour of the extracted CFG, i.e. $b(\mathcal{G}_{\text{JBC}})$, in the standard sense of simulation between labelled transition systems [29]. In general, this requires to show that every initial configuration of P_{jbc} is simulated by some initial configuration of $b(\mathcal{G}_{\text{JBC}})$. In our case there is a single initial program configuration:

$$c_{init} = (\langle main, 0, f, \epsilon, z\rangle; h)$$

The required simulation can be established by exhibiting a concrete *simulation relation* between JVM configurations and CFG behavioural configurations that relates the initial P_{jbc}-configuration c_{init} to an initial $b(\mathcal{G}_{\text{JBC}})$-state.

We prove that the abstraction function θ, viewed as a relation, is such a simulation relation. First, we observe that $\theta(c_{init}) = (v^0_{main}, \epsilon)$ is an initial state of $b(\mathcal{G}_{\text{JBC}})$, see Definition 3. Then, we prove that for any JVM configuration c and any transition $c \xrightarrow{l} c'$ in the execution of P_{jbc} there is a matching transition $\theta(c) \xrightarrow{l} \theta(c')$ in $b(\mathcal{G}_{\text{JBC}})$. We show this by case analysis on the top frame $\langle m, p, f, s, z\rangle$ or $\langle x\rangle_{\text{exc}}$, and in the former case also on the type of the instruction $m[p]$, i.e. the current JBC instruction. For each case we deduce the possible labelled transitions $c \xrightarrow{l} c'$ from c as induced by the JBC operational semantics rules shown on Fig. 5. Next, we use the CFG construction rules from Fig. 6 to determine the nodes and edges extracted for instruction $m[p]$, and the definition of CFG behaviour (Definition 3) to establish $\theta(c) \xrightarrow{l} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $i = m[p] \in$ CmpInst Applying the abstraction function on configura-

tion c we have: $\theta(c) = (\circ_m^p, \sigma)$ where σ denotes $\gamma(A)$. The operational semantics of [CMP] defines that there is a next configuration c' such that: $c \xrightarrow{\tau} c'$ where $c' = \big(\langle m, succ(p), f', s', z'\rangle \cdot A; h'\big)$. From the definition of the CFG extraction function $\mathcal{G}_{\text{JBC}}$ we have $(\circ_m^p, \varepsilon, \circ_m^{succ(p)}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$. By the definition of CFG behaviour this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{\tau}_b (\circ_m^{succ(p)}, \sigma)$. Now applying the abstraction function on c' we obtain: $\theta(c') = (\circ_m^{succ(p)}, \sigma)$ and therefore: $\theta(c) \xrightarrow{\tau} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $i = m[p] \in$ CNDINST Applying the abstraction function on configuration c we have: $\theta(c) = (\circ_m^p, \sigma)$ where σ denotes $\gamma(A)$. The operational semantics of [CND] defines that there is a next configuration c' such that: $c \xrightarrow{\tau} c'$. Then there are two sub-cases:

1. $cond(\langle m, p, f, s, z\rangle) = true$: Then $c' = (\langle m, jmp(i), f, s', z\rangle \cdot A; h)$. From the definition of the CFG extraction function $\mathcal{G}_{\text{JBC}}$ we have $(\circ_m^p, \varepsilon, \circ_m^{jmp(i)}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$. By the definition of CFG behaviour this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{\tau}_b (\circ_m^{jmp(i)}, \sigma)$. Now applying the abstraction function on c' we obtain: $\theta(c') = (\circ_m^{jmp(i)}, \sigma)$ and therefore: $\theta(c) \xrightarrow{\tau} \theta(c')$.
2. $cond(\langle m, p, f, s, z\rangle) = false$: Then $c' = (\langle m, succ(p), f, s', z\rangle \cdot A; h)$. From the CFG extraction function we have $(\circ_m^p, \varepsilon, \circ_m^{succ(p)}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$. By the definition of CFG behaviour this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{\tau}_b (\circ_m^{succ(p)}, \sigma)$. Now applying the abstraction function on c' we obtain: $\theta(c') = (\circ_m^{succ(p)}, \sigma)$ and therefore: $\theta(c) \xrightarrow{\tau} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $i = m[p] \in$ JMPINST Applying the abstraction function on configuration c we have: $\theta(c) = (\circ_m^p, \sigma)$ where σ denotes $\gamma(A)$. The operational semantics of [JMP] defines that there is a next configuration c' such that: $c \xrightarrow{\tau} c'$ where $c' = (\langle m, jmp(i), f, s, z\rangle \cdot A; h)$. From the definition of the CFG extraction function $\mathcal{G}_{\text{JBC}}$ we have $(\circ_m^p, \varepsilon, \circ_m^{jmp(i)}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$. By the definition of CFG behaviour, this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{\tau}_b (\circ_m^{jmp(i)}, \sigma)$. Now applying the abstraction function on c' we obtain: $\theta(c') = (\circ_m^{jmp(i)}, \sigma)$ and therefore: $\theta(c) \xrightarrow{\tau} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $i = m[p] \in$ THRINST The single instruction in the set is `throw X`. Let X be the set containing the static type of the exception being thrown, and all of its subtypes. Applying the abstraction function on configuration c we obtain: $\theta(c) = (\circ_m^p, \sigma)$ where σ denotes $\gamma(A)$. The operational semantics of `throw` X defines that for every exception $x \in X$ there is a next configuration c' such that: $c \xrightarrow{throw\ x} c'$ where $c' = (\langle x\rangle_{\texttt{exc}} \cdot \langle m, p, f, s, z\rangle \cdot A; h)$. Then there are two sub-cases:

1. Method m has a handler for x (i.e. $k_{\Gamma[m]}^{p,x} = t \ \wedge \ t \neq 0$). From the definition of the CFG construction function $\mathcal{G}_{\text{JBC}}$, we have: $(\circ_m^p, \varepsilon, \bullet_m^{p,x}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$.
By the definition of CFG behaviour, this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{throw\ x}_b (\bullet_m^{p,x}, \sigma)$.
Now, since $k_{\Gamma[m]}^{p,x} \neq 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet_m^{p,x}, \sigma)$ and therefore: $\theta(c) \xrightarrow{throw\ x} \theta(c')$.
2. There is no handler in m for x (i.e. $k_{\Gamma[m]}^{p,x} = 0$). From the CFG extraction function we obtain: $(\circ_m^p, \varepsilon, \bullet_m^{p,x,r}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$. By the definition of CFG behaviour, we have the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{throw\ x}_b (\bullet_m^{p,x,r}, \sigma)$.
Then since $k_{\Gamma[m]}^{p,x} = 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet_m^{p,x,r}, \sigma)$ and therefore again: $\theta(c) \xrightarrow{throw\ x} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $i = m[p] \in$ XMPINST Applying the abstraction function on configuration c we have: $\theta(c) = (\circ_m^p, \sigma)$ where σ denotes $\gamma(A)$. Then there are two sub-cases:

1. $\nu(\langle m, p, f, s, z\rangle, h) = undef$: The operational semantics of [XMP] defines that there is a next configuration c' such that: $c \xrightarrow{\tau} c'$ where $c' = \big(\langle m, succ(p), f', s', z'\rangle \cdot A; h'\big)$. From the definition of the CFG extraction function $\mathcal{G}_{\text{JBC}}$ we have $(\circ_m^p, \varepsilon, \circ_m^{succ(p)}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$. By the definition of CFG behaviour, this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{\tau}_b (\circ_m^{succ(p)}, \sigma)$. Now applying the abstraction function on c' we obtain: $\theta(c') = (\circ_m^{succ(p)}, \sigma)$ and therefore: $\theta(c) \xrightarrow{\tau} \theta(c')$.
2. $\nu(\langle m, p, f, s, z\rangle, h) = x$: The operational semantics of [XMP] defines that there exists a next configuration c' such that: $c \xrightarrow{throw\ x} c'$ where $c' = (\langle x\rangle_{\texttt{exc}}\langle \cdot m, p, f, s, z\rangle \cdot A; h)$. Again there are two sub-cases:
   (a) Method m has a handler for x (i.e. $k_{\Gamma[m]}^{p,x} = t \wedge t \neq 0$). From the CFG construction function definition, we have: $(\circ_m^p, \varepsilon, \bullet_m^{p,x}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$.
   By the definition of CFG behaviour, this edge entails the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{throw\ x}_b (\bullet_m^{p,x}, \sigma)$. Now, since $k_{\Gamma[m]}^{p,x} \neq 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet_m^{p,x}, \sigma)$ and therefore: $\theta(c) \xrightarrow{throw\ x} \theta(c')$.
   (b) There is no handler in m for x (i.e. $k_{\Gamma[m]}^{p,x} = 0$). From the CFG extraction function we obtain: $(\circ_m^p, \varepsilon, \bullet_m^{p,x,r}) \in \mathcal{G}_{\text{JBC}}^{m,p,i}$.
   By the definition of CFG behaviour, we have the behavioural transition: $(\circ_m^p, \sigma) \xrightarrow{throw\ x}_b (\bullet_m^{p,x,r}, \sigma)$. Then since $k_{\Gamma[m]}^{p,x} = 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet_m^{p,x,r}, \sigma)$ and therefore again: $\theta(c) \xrightarrow{throw\ x} \theta(c')$.

Case $c = (\langle x\rangle_{\texttt{exc}} \cdot \langle m, p, f, s, z\rangle \cdot A; h)$ Recall that when there is an exception record on the stack, the JVM takes control, and execution continues with an internal transition. Based on the exception handling table in m we have again two sub-cases:

1. Method m has a handler for x ($k^{p,x}_{\Gamma[m]} = t \ \wedge \ t \neq 0$). In this case, applying the abstraction function on the current configuration we obtain: $\theta(c) = (\bullet^{p,x}_m, \sigma)$ where σ again denotes $\gamma(A)$. According to the operational semantics of JVM we have the transition: $c \xrightarrow{catch\ x} c'$ where $c' = \big(\langle m, t, f, s, z\rangle \cdot A; h'\big)$. From the CFG extraction function for any instruction that can raise exception x we have: $(\bullet^{p,x}_m, \varepsilon, \circ^t_m) \in \mathcal{G}^{m,p,i}_{\text{JBC}}$. Based on the definition of CFG behaviour, this edge induces the transition: $(\bullet^{p,x}_m, \sigma) \xrightarrow{catch\ x}_b (\circ^t_m, \sigma)$. Applying the abstraction function on c' we have: $\theta(c') = (\circ^t_m, \sigma)$ and hence: $\theta(c) \xrightarrow{catch\ x} \theta(c')$.
2. There is no handler in m for x ($k^{p,x}_{\Gamma[m]} = 0$). Applying the abstraction function on the current configuration we obtain: $\theta(c) = (\bullet^{p,x,r}_m, \sigma)$. Let $A = \langle n, q, f, s, z\rangle \cdot A'$. Then n is the caller of m, and by the operational semantics of JVM we have: $c \xrightarrow{m\ xret\ n} c'$ where $c' = (\langle x\rangle_{\texttt{exc}} \cdot \langle n, q, f, s, z\rangle \cdot A'; h')$. From the CFG extraction function for any instruction $m[p]$ that can raise exception x we obtain: $(\circ^p_m, \varepsilon, \bullet^{p,x,r}_m) \in \mathcal{G}^{m,p,i}_{\text{JBC}}$. Because there is no handler in m, the propagated exception x will be in the CFG extracted for n. So, based on the exception handling table in n we in turn have the following two sub-cases:
   - Method n has a handler for x ($k^{q,x}_{\Gamma[n]} = t \ \wedge \ t \neq 0$). Then according to the CFG extraction rules for n: $(\circ^q_n, m, \bullet^{q,x}_n) \in \mathcal{G}_{\text{JBC}}(n)$. By the definition of CFG behaviour: $(\bullet^{p,x,r}_m, \circ^q_n \cdot \sigma') \xrightarrow{m\ xret\ n}_b (\bullet^{q,x}_n, \sigma')$. Since $k^{q,x}_{\Gamma[n]} \neq 0$, abstracting c' we obtain: $\theta(c') = (\bullet^{q,x}_n, \sigma')$ and therefore: $\theta(c) \xrightarrow{m\ xret\ n} \theta(c')$.
   - Method n does not define any handler for x ($k^{q,x}_{\Gamma[n]} = 0$). Then according to the CFG extraction rules for n: $(\circ^q_n, m, \bullet^{q,x,r}_n) \in \mathcal{G}_{\text{JBC}}(n)$. By the definition of CFG behaviour: $(\bullet^{p,x,r}_m, \circ^q_n \cdot \sigma') \xrightarrow{m\ xret\ n}_b (\bullet^{q,x,r}_n, \sigma')$. Since $k^{q,x}_{\Gamma[n]} = 0$, abstracting c' we obtain: $\theta(c') = (\bullet^{q,x,r}_n, \sigma')$ and hence: $\theta(c) \xrightarrow{m\ xret\ n} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $i = m[p] \in$ InvInst Applying the abstraction function on configuration c we have: $\theta(c) = (\circ^p_m, \sigma)$ where σ denotes $\gamma(A)$. There are two sub-cases:

1. $\nu(\langle m, p, f, s, z\rangle, h) = \texttt{N.P.E.}$: The operational semantics of [Inv] defines that there is a next configuration c' such that: $c \xrightarrow{throw\ \texttt{N.P.E.}} c'$ where $c' = (\langle \texttt{N.P.E.}\rangle_{\texttt{exc}} \cdot \langle m, p, f, s, z\rangle \cdot A; h')$. The rest is proved in two sub-cases:
   (a) Method m has a handler for N.P.E. (i.e. $k^{p,\texttt{N.P.E.}}_{\Gamma[m]} = t \wedge t \neq 0$). From the CFG construction function we have: $(\circ^p_m, \varepsilon, \bullet^{p,\texttt{N.P.E.}}_m) \in \mathcal{G}^{m,p,i}_{\text{JBC}}$. By the definition of CFG behaviour, this edge entails the behavioural transition: $(\circ^p_m, \sigma) \xrightarrow{throw\ \texttt{N.P.E.}}_b (\bullet^{p,\texttt{N.P.E.}}_m, \sigma)$. Now, since $k^{p,\texttt{N.P.E.}}_{\Gamma[m]} \neq 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet^{p,\texttt{N.P.E.}}_m, \sigma)$ and therefore: $\theta(c) \xrightarrow{throw\ \texttt{N.P.E.}} \theta(c')$.
   (b) There is no handler in m for x (i.e. $k^{p,\texttt{N.P.E.}}_{\Gamma[m]} = 0$). From the CFG extraction function we obtain: $(\circ^p_m, \varepsilon, \bullet^{p,\texttt{N.P.E.},r}_m) \in \mathcal{G}^{m,p,i}_{\text{JBC}}$. By the definition of CFG behaviour, we have: $(\circ^p_m, \sigma) \xrightarrow{throw\ \texttt{N.P.E.}}_b (\bullet^{p,\texttt{N.P.E.},r}_m, \sigma)$. Then since $k^{p,\texttt{N.P.E.}}_{\Gamma[m]} = 0$, applying the abstraction function on c' we obtain: $\theta(c') = (\bullet^{p,\texttt{N.P.E.},r}_m, \sigma)$ and therefore again: $\theta(c) \xrightarrow{throw\ \texttt{N.P.E.}} \theta(c')$.
2. $\nu(\langle m, p, f, s, z\rangle, h) \neq \texttt{N.P.E.}$: In this case the control will pass to the callee method and after completion, it will return back to the caller context and next instruction will be fetched. The possible receivers (one edge for each receiver) are determined by the virtual method call resolution, if the instruction is virtual invocation. The operational semantics of [Inv] defines that there is a next configuration c' such that: $c \xrightarrow{m\ call\ n} c'$ where $c' = \big(\langle n, 0, f', \epsilon, z'\rangle \cdot \langle m, p, f, s, z\rangle \cdot A; h'\big)$. From the definition of the CFG extraction function $\mathcal{G}_{\text{JBC}}$ for all n in the set of the possible receivers resulted by call resolution algorithm we have: $(\circ^p_m, n, \circ^{succ(p)}_m) \in \mathcal{G}^{m,p,i}_{\text{JBC}}$. By the definition of CFG behaviour, this edge entails the behavioural transition: $(\circ^p_m, \sigma) \xrightarrow{m\ call\ n}_b (\circ^0_n, \circ^p_m \cdot \sigma)$. Now applying the abstraction function on c' we obtain: $\theta(c') = (\circ^0_n, \sigma')$. If we take $\sigma' = \circ^p_m \cdot \sigma$ then we can prove: $\theta(c) \xrightarrow{m\ call\ n} \theta(c')$.

Case $c = (\langle m, p, f, s, z\rangle \cdot A; h)$ and $m[p] \in$ RetInst Applying the abstraction function on configuration c we have: $\theta(c) = (\circ^{p,r}_m, \sigma)$ where σ denotes $\gamma(A)$. Let $A = \langle n, q, f', s', z'\rangle \cdot A'$. Then n is the caller of m, and by the operational semantics of [Ret] there is a next configuration c' such that: $c \xrightarrow{m\ ret\ n} c'$ where $c' = (\langle n, succ(q), f', s'', z'\rangle \cdot A'; h)$. Then according to the CFG extraction rules for n: $(\circ^q_n, m, \circ^{succ(q)}_n) \in \mathcal{G}^{n,q}_{\text{JBC}}$. By the definition of CFG behaviour: $(\circ^{p,r}_m, \circ^q_n \cdot \sigma') \xrightarrow{m\ ret\ n}_b (\circ^{succ(q)}_n, \sigma')$. Abstracting next configuration c' we obtain: $\theta(c') = (\circ^{succ(q)}_n, \sigma')$ and therefore: $\theta(c) \xrightarrow{m\ ret\ n} \theta(c')$. □

10 Appendix: Weak simulation on models

The results presented here follow closely Milner [29], adapted to initialized models and thus to CFGs. As usual, in the context of a transition relation $\rightarrow$, we shall write $p_i \xrightarrow{l} p_j$ to denote $(p_i, l, p_j) \in \rightarrow$. Also, we use ε to label silent edges (instead of τ used in Milner [29], to be consistent with our notation for models). The transition relation $\rightarrow$ induces a *weak* labelled transition relation in the standard fashion, where $\beta \neq \varepsilon$:

$$\Longrightarrow \stackrel{def}{=} \xrightarrow{\varepsilon}{}^{\star}$$
$$\stackrel{\beta}{\Longrightarrow} \stackrel{def}{=} \Longrightarrow \xrightarrow{\beta} \Longrightarrow$$

The notion of *weak simulation* on models is based on the standard notion, but requires an additional agreement on the atomic propositions.

Definition 8 (*Weak Simulation on Models*) Let $(\mathcal{M}_p, \mathbb{E}_p)$ and $(\mathcal{M}_q, \mathbb{E}_q)$ be two initialized models, with $\mathcal{M}_p = (S_p, L, \rightarrow_p, A, \lambda_p)$ and $\mathcal{M}_q = (S_q, L, \rightarrow_q, A, \lambda_q)$, and let $R \subseteq V_p \times V_q$. Then R is a *weak simulation* if for all $(p_i, q_i) \in R$ the following conditions hold:

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

We say that q weakly simulates p if $(p, q) \in R$ for *some* weak simulation relation R. We also say that $(\mathcal{M}_q, \mathbb{E}_q)$ weakly simulates $(\mathcal{M}_p, \mathbb{E}_p)$ if for every $p \in \mathbb{E}_p$ there is $q \in \mathbb{E}_q$ such that q weakly simulates p.

The following proposition (again the style of [29]) allows for more compact simulation proofs.

Proposition 1 *Let $(\mathcal{M}_p, \mathbb{E}_p)$ and $(\mathcal{M}_q, \mathbb{E}_q)$ be two initialized models, with $\mathcal{M}_p = (S_p, L, \rightarrow_p, A, \lambda_p)$ and $\mathcal{M}_q = (S_q, L, \rightarrow_q, A, \lambda_q)$, and let $R \subseteq V_p \times V_q$. Then R is a* weak simulation *if for all $(p_i, q_i) \in R$ the following conditions hold:*

1. *$\lambda_p(p_i) = \lambda_q(q_i)$;*
2. *if $p_i \xrightarrow{\varepsilon} p_j$ then there is $q_j \in V_q$ such that $q_i \Longrightarrow q_j$ and $(p_j, q_j) \in R$;*
3. *if $p_i \xrightarrow{\beta} p_j$ then there is $q_j \in V_q$ such that $q_i \stackrel{\beta}{\Longrightarrow} q_j$ and $(p_j, q_j) \in R$.*

Thus, to prove weak simulation, it suffices to show that every strong transition of the first model is matched by a correspondingly labelled weak transition of the second model.

11 Appendix: Correctness of $\mathcal{G}_{\mathbf{BIR}} \circ$ `BC2BIR`

The `BC2BIR` transformation may collapse several bytecode instructions into a single BIR instruction. We divide the JBC instructions into two sets: the *producer* instructions, i.e., those that produce at least one BIR instruction in function `BC2BIR`$_{instr}$, and the *auxiliary* ones, i.e., those that produce none. This division can be deduced from Fig. 9 (on page 22). 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 sub sequences delimited by producer instructions. Thus each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Equivalently, we may say that a segment is defined by an optional *basic block* [1] and a subsequent producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java Bytecode Verifier (see Sect. 2.1). All methods in such programs must have `goto`, `return`, or `athrow` as the last (reachable) instruction, which are producer instructions. Therefore, there can not be contiguous (reachable) instructions that are not delimited by a producer instruction.

Each bytecode segment is transformed into a set of contiguous instructions by `BC2BIR`. We call this set a *BIR segment*, which is a partition of the BIR instruction array. There exists a one-to-one mapping between bytecode segments and the BIR segments, which is also order-preserving. Thus, we can associate each instruction, either in the JBC or BIR arrays, to the unique index of its correspondent bytecode segment. Figure 10 (on page 23) illustrates the partitioning of instructions into segments. Method `odd` has eight bytecode (and BIR) segments, as indicated by the distinct shades. Producer instructions are underlined.

In the definition of the direct extraction algorithm in Fig. 6 (on page 13), 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-like graph of internal transfer edges of the same length as the number of auxiliary instructions, followed by the edges generated for the producer instruction. Let p be the position for the first auxiliary instruction, and q the position of the producer instruction. We illustrate the pattern for this path-like graph below.

$$\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)} \xrightarrow{\varepsilon} \circ_m^{succ(succ(p))} \xrightarrow{\varepsilon} \cdots \circ_m^q$$

It is easy to see that the path-like graph is weakly simulated by some reflexive edge $\circ_m^{\mathrm{pc}} \Longrightarrow \circ_m^{\mathrm{pc}}$. Therefore, for simplicity we present the proof for the case where $p = q$. That is, for JBC segments without auxiliary instructions.

Another important observation is about the mapping between control addresses between the JBC and the BIR

representations. A control address q from a branching instruction (e.g., `goto q` or `ifeq q`) or from an exception handler is always mapped to the first control address (let's call it `pc`) of the corresponding BIR segment that q belongs to. This is necessarily the case because either q contains a producer instruction, which will generate a set of sequential BIR instructions, with smallest control address being `pc`, or it contains an auxiliary instruction, which will be collapsed into the BIR instructions when the first JBC producer instruction is processed, also having `pc` as the control address with smallest index.

Based on the observations above, our main theorem states that the method graphs extracted using the indirect algorithm weakly simulate (see Definition 8) the method graphs using the direct algorithm. The abstract stacks are omitted in the proof since examining the instructions is sufficient to produce the edges.

We now enunciate Theorem 2 again, and present the full proof.

Theorem (Structural Simulation of CFGs) *Let Γ be the environment modeling a well-formed JBC program P_{jbc}. Then* $(\mathcal{G}_{\text{BIR}} \circ \texttt{BC2BIR})(\Gamma)$ *weakly simulates* $\mathcal{G}_{\text{JBC}}(\Gamma)$.

Proof Let m be a method signature and $\Gamma[m]$ be the method's definition in Γ. Let p range over indices in the bytecode instructions array, `pc` over indices in the BIR instructions array, $\circ_m^{p,x,r}$ over control nodes in $\mathcal{G}_{\text{JBC}}(\Gamma[m])$, and $\circ_m^{\texttt{pc},x,r}$ over control nodes in $(\mathcal{G}_{\text{BIR}} \circ \texttt{BC2BIR})(\Gamma[m])$. The control nodes are valuated with two optional atomic propositions: x, which is an exception type, and r, which is the atomic proposition denoting a return point. Further, let $seg_{JBC}(m, p)$ and $seg_{BIR}(m, \texttt{pc})$ be two auxiliary functions that return the segment index that a JBC, or a BIR control address belongs to, respectively. Let s be the index of a BIR segment. Function $fst(s)$ return its first control address and $oap(s, XR)$ return the set of control addresses in s tagging a node with the non-empty set of *optional atomic propositions* equal to XR.

We define the binary relation R as the union of two binary relations R_1 and R_2, as follows:

$$R \stackrel{def}{=} R_1 \cup R_2$$

$$R_1 \stackrel{def}{=} \{(\circ_m^p, \circ_m^{\texttt{pc}}) \mid seg_{JBC}(m, p) = seg_{BIR}(m, \texttt{pc}) \wedge \texttt{pc} = fst(seg_{BIR}(m, \texttt{pc}))\}$$

$$R_2 \stackrel{def}{=} \{(\circ_m^{p,XR}, \circ_m^{\texttt{pc},XR}) \mid seg_{JBC}(m, p) = seg_{BIR}(m, \texttt{pc}) \wedge \texttt{pc} \in oap(seg_{BIR}(m, \texttt{pc}), XR)\}$$

and show the relation to be a weak simulation in the standard fashion, following Proposition 1: for every pair of nodes in R, we first show that the nodes have the same set of atomic propositions, and then match every strong edge that has the first node as source, to a corresponding weak edge that has the second node as source, so that the target nodes are again related by R.

Intuitively, the direct and indirect algorithms extract a similar branching structure for the same JBC code segment, differing in the occurrences of silent transitions. Therefore, R relates the first normal (source) nodes extracted by both algorithms (i.e., R_1), and a node from the direct algorithm tagged with a non-empty set of atomic propositions to nodes extracted in the indirect algorithm with the same set of atomic propositions (i.e., R_2). Notice that the only case where the indirect algorithm produces two distinct nodes with the same set of non-empty atomic propositions is for the case of method invocations, where a `N.P.E.` may be raised in different control addresses: either by a `[notnull]` or propagated by the callee method.

Let $(\circ_m^p, \star) \in R_1$ and let $(\circ_m^{p,r}, \star) \in R_2$ where $\circ_m^p$ and $\circ_m^{p,r}$ are control nodes in $\mathcal{G}_{\text{JBC}}(\Gamma[m])$ and $\star$ is a control node in $(\mathcal{G}_{\text{BIR}} \circ \texttt{BC2BIR})(\Gamma[m])$. We consider the two cases separately. Let first $(\circ_m^p, \star) \in R_1$. The proof proceeds by *case analysis* on the *type of the producer instruction* of the bytecode segment $seg_{JBC}(m, p)$ giving rise to $\circ_m^p$. The cases follow the subsets of JBC instructions presented in Fig. 4, which share the same extraction rule in the direct algorithm for its instructions. We rely on the notation to indicate that two nodes have the same set of atomic propositions.

Case $i \in$ CmpInst There are two producer instructions in this subset: `nop` and `store`. We present the case for `store` only, since it subdivides into two sub-cases, the first of which is analogous to the case of `nop`.

The direct extraction always produces a single edge from one normal node to the node tagged with the successor of p in the instructions array:

$$\mathcal{G}_{\text{JBC}}^{m,p,\texttt{store}} = \{\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)}\}$$

The BIR transformation can return either one or two assignments, which leads to two subcases in the proof.

Subcase I $\texttt{BC2BIR}_{instr}$ produces a single assignment for `store`. Applying the *Assignment* rule of $\mathcal{G}_{\text{BIR}}$, we obtain a single edge:

$$\texttt{BC2BIR}_{instr}^{p,\texttt{store}} = \{[\texttt{l}_\texttt{j}\texttt{:=e}]\}$$

$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{l}_\texttt{j}\texttt{:=e}]} = \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}\}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = fst(s)$. There is the edge $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)}$, which is matched by the weak edge $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc+1}}$. Moreover, also $(\circ_m^{succ(p)}, \circ_m^{\texttt{pc+1}}) \in R$ since $seg_{JBC}(m, succ(p)) = seg_{BIR}(m, \texttt{pc+1})$ and $\texttt{pc+1} = fst(seg_{BIR}(m, \texttt{pc+1}))$. That is, `pc+1` is the first control address in the next code segment. The case for `nop` is analogous to this.

Subcase II $\texttt{BC2BIR}_{instr}$ produces two assignments. Applying the *Assignment* rule from $\mathcal{G}_{\text{BIR}}$ twice, we have:

$$\texttt{BC2BIR}_{instr}^{p,\texttt{store}} = \{[\texttt{t}_\texttt{k}{:=}\texttt{l}_\texttt{j}]; [\texttt{l}_\texttt{j}{:=}\texttt{e}]\}$$
$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{t}_\texttt{k}:=\texttt{l}_\texttt{j}]} = \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}\}$$
$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc+1},[\texttt{l}_\texttt{j}:=\texttt{e}]} = \{\circ_m^{\texttt{pc+1}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+2}}\}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = \mathit{fst}(s)$. There is an edge $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)}$. It is matched by the weak edge $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc+2}}$, which traverses $\circ_m^{\texttt{pc+1}}$. Then, also $(\circ_m^{succ(p)}, \circ_m^{\texttt{pc+2}}) \in R$ since $seg_{JBC}(m, succ(p)) = seg_{BIR}(m, \texttt{pc+2})$ and $\texttt{pc+2} = \mathit{fst}(seg_{BIR}(m, \texttt{pc+2}))$. That is, $\texttt{pc+2}$ is the first control address in the next code segment.

Case $i \in$ CndInst The only producer instruction in this subset is $\texttt{if}\ q$. The direct extraction produces two edges from the normal node tagged with control address p: one to the node tagged with the successor control address to p, and another to the node tagged with the branching control address q:

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

The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction, applied to which $\mathcal{G}_{\text{BIR}}$ produces two edges:

$$\texttt{BC2BIR}_{instr}^{p,\texttt{if}\ q} = [\texttt{if}\mathit{expr}\ \texttt{pc}']$$
$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{if}\ expr\ \texttt{pc}']} = \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}'}\}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = \mathit{fst}(s)$. The first edge $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)}$ is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc+1}}$. Moreover, $(\circ_m^{succ(p)}, \circ_m^{\texttt{pc+1}}) \in R$ since $seg_{JBC}(m, succ(p)) = seg_{BIR}(m, \texttt{pc+1})$ and $\texttt{pc+1} = \mathit{fst}(seg_{BIR}(m, \texttt{pc+1}))$. The second edge $\circ_m^p \xrightarrow{\varepsilon} \circ_m^q$ is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc}'}$, and again $(\circ_m^q, \circ_m^{\texttt{pc}'}) \in R$ since $seg_{JBC}(m, q) = seg_{BIR}(m, \texttt{pc}')$ and $\texttt{pc}' = \mathit{fst}(seg_{BIR}(m, \texttt{pc}'))$. That is, both $\texttt{pc+1}$ and $\texttt{pc}'$ are the first control addresses in their respective code segments.

Case $i \in$ JmpInst The only producer instruction in this subset is $\texttt{goto}\ q$. The direct extraction produces a single edge from one normal node to another normal node tagged with the branching control address q:

$$\mathcal{G}_{\text{JBC}}^{m,p,\texttt{goto}\ q} = \{\circ_m^p \xrightarrow{\varepsilon} \circ_m^q\}$$

The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction, applied to which $\mathcal{G}_{\text{BIR}}$ produces a single edge:

$$\texttt{BC2BIR}_{instr}^{p,\texttt{goto}\ q} = [\texttt{goto pc}']$$
$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{goto pc}']} = \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}'}\}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = \mathit{fst}(s)$. There is the edge $\circ_m^p \xrightarrow{\varepsilon} \circ_m^q$ which is matched by the corresponding edge $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc}'}$. Moreover, $(\circ_m^q, \circ_m^{\texttt{pc}'}) \in R$ since $seg_{JBC}(m, q) = seg_{BIR}(m, \texttt{pc}')$ and $\texttt{pc}' = \mathit{fst}(seg_{BIR}(m, \texttt{pc}'))$. That is, $\texttt{pc}'$ is the first control address on its code segment.

Case $i \in$ ThrInst Let $\texttt{X}$ be the set of all possible exception types for an instance of the idealized $\texttt{throw X}$, which is the single instruction in ThrInst. Then $\texttt{X} \subseteq \{x | x <: \mathit{staticT}(\texttt{e})\}$ for the corresponding instance of $[\texttt{throw e}]$. That is, the set of possible exceptions in the indirect extraction soundly over-approximates the set $\texttt{X}$ since any exception $x \in \texttt{X}$ is necessarily a sub-type of $\mathit{staticT}(\texttt{e})$. Let $x \in \texttt{X}$.

The direct extraction for the $\texttt{throw}$ instruction produces two edges if there is a suitable handler for the exception x in position p. Otherwise, it produces a single edge, whose sink node is an exceptional return node:

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

The transformation $\texttt{BC2BIR}_{instr}$ returns a single instruction. Then, similarly to $\mathcal{G}_{\text{JBC}}$, the $\mathcal{G}_{\text{BIR}}$ function produces either one, or two edges:

$$\texttt{BC2BIR}_{instr}^{p,\texttt{athrow}} = [\texttt{throw e}]$$
$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{throw e}]} = \begin{cases} \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc},x}, \bullet_m^{\texttt{pc},x} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}'}\} & \text{if has handler} \\ \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc},x,r}\} & \text{otherwise} \end{cases}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = \mathit{fst}(s)$. In the case where there is an exception handler for x in p and $\texttt{pc}$, the edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x}$ is matched by the corresponding weak edge $\circ_m^{\texttt{pc}} \Longrightarrow \bullet_m^{\texttt{pc},x}$. Then also $(\bullet_m^{p,x}, \bullet_m^{\texttt{pc},x}) \in R$ since $seg_{JBC}(p) = seg_{BIR}(m, \texttt{pc})$ and $\texttt{pc} \in oap(s, \{x\})$. That is, $\texttt{pc}$ tags a node where the set $XP = \{x\}$. Actually, for this case there is only one node since $\texttt{pc}$ is the only control address in the segment. Further, there is the edge $\bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^q$, which is matched by $\bullet_m^{\texttt{pc},x} \Longrightarrow \circ_m^{\texttt{pc}'}$, and again $(\circ_m^q, \circ_m^{\texttt{pc}'}) \in R$ since $seg_{JBC}(m, q) = seg_{BIR}(m, \texttt{pc}')$ and $\texttt{pc}' = \mathit{fst}(seg_{BIR}(m, \texttt{pc}'))$. That is, $\texttt{pc}'$ is the first control address on its code segment.

In the case where there is no exception handler for x, the only edge is $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x,r}$, which is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \bullet_m^{\texttt{pc},x,r}$. Moreover, $(\bullet_m^{p,x,r}, \bullet_m^{\texttt{pc},x,r}) \in R$ since $seg_{JBC}(p) = seg_{BIR}(m, \texttt{pc})$ and $\texttt{pc} \in oap(s, \{x, r\})$. That is, $\texttt{pc}$ tags the node where the set $XP = \{x, r\}$, which concludes the case.

Case $i \in$ 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. We present this case for the $\texttt{div}$ instruction, which

can only raise x = `ArithmeticException`. The cases for all other instructions in XMPINST are analogous.

The rule for the direct extraction produces one normal edge, for the case of successful execution. Also, for each exception that the instruction may raise, it outputs a varying number of edges: a pair if there is a suitable handler for the exception, and otherwise a single edge:

$$\mathcal{G}_{\text{JBC}}^{m,p,\texttt{div}} = \begin{cases} \{\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)},\ \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x},\ \bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^q\} & \text{if has handler} \\ \{\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)},\ \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x,r}\} & \text{otherwise} \end{cases}$$

The $\texttt{BC2BIR}_{instr}$ transformation returns a single instruction, which is an assertion. The $\mathcal{G}_{\text{BIR}}$ function always produces one edge to a normal node, denoting normal execution. Also, it outputs a varying number of edges: two edges, if there is a suitable exception handler; otherwise it outputs a single edge to an exceptional return node. Thus we may have two sets of edges:

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

$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{notzero}]} = \begin{cases} \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc},x}, \bullet_m^{\texttt{pc},x} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}'}\} & \text{if has handler} \\ \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc},x,r}\} & \text{otherwise} \end{cases}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = fst(s)$. The edge $\circ_m^p \xrightarrow{\varepsilon} \circ_m^{succ(p)}$ is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc+1}}$ in both cases. Moreover, also $(\circ_m^{succ(p)}, \circ_m^{\texttt{pc+1}}) \in R$ since $seg_{JBC}(m, succ(p)) = seg_{BIR}(m, \texttt{pc+1})$ and $\texttt{pc+1} = fst(seg_{BIR}(m, \texttt{pc+1}))$. That is, pc+1 is the first control address in the next segment.

In the case where there is an exception handler for x, there is the edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x}$, which is matched by $\circ_m^{\texttt{pc}} \overset{\varepsilon}{\Longrightarrow} \bullet_m^{\texttt{pc},x}$. Thus, also $(\bullet_m^{p,x}, \bullet_m^{\texttt{pc},x}) \in R$ $seg_{JBC}(m, p) = seg_{BIR}(m, \texttt{pc})$ and $\texttt{pc} \in oap(s, \{x\})$. Similarly to THRINST, in this case pc is the only control address tagging a node also tagged by x since it is the only address in the segment. Also, there is the edge $\bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^q$, which is matched by $\bullet_m^{\texttt{pc},x} \Longrightarrow \circ_m^{\texttt{pc}'}$, and again $(\circ_m^q, \circ_m^{\texttt{pc}'}) \in R$ since $seg_{JBC}(m, q) = seg_{BIR}(m, \texttt{pc}')$ and $\texttt{pc}' = fst(seg_{BIR}(m, \texttt{pc}'))$.

If there is no exception handler for x, then the direct algorithm produces the edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x,r}$, which is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \bullet_m^{\texttt{pc},x,r}$. Then, $(\bullet_m^{p,x,r}, \bullet_m^{\texttt{pc},x,r}) \in R$ since $seg_{JBC}(p) = seg_{BIR}(m, \texttt{pc})$ and $\texttt{pc} \in oap(s, \{x, r\})$. That is, pc tags the node where the set $XP = \{x, r\}$, which concludes the case.

Case $i \in$ INVINST This is the subset of instructions that execute method invocations: `invokespecial`, `invokestatic`, `invokevirtual`, and `invokeinterface`. The first two instructions always have a single receiver for the invocation, while the last two instruction may have more than one potential receiver, because of the dynamic dispatching.

This is reflected in the function Rec_{Γ}^{i} in Fig. 6, which uses a virtual method call algorithm α to list the possible receivers for a method invocation. The indirect algorithm has an equivalent function, res^{α}, which returns the single receiver for a non-virtual method call, or uses the same algorithm α to list the possible receivers of a virtual call. The set of possible receivers for a method invocation is the same for both algorithms. Therefore it suffices to show that, for a (potential) receiver of a method invocation, the subgraph extracted indirectly weakly simulates the subgraph extracted directly.

We present the proof for the single receiver of an `invokespecial` call. We chose this instruction because it is the only one that may produce two distinct sets of BIR instructions (see Fig. 9). We thus consider two subcases: (*I*) when the receiver is an object constructor, and (*II*), when it is a private method or a method from the super class. For the latter, the BIR instructions are the same as for `invokestatic`, `invokevirtual`, and `invokeinterface`, and the proofs are analogous.

The direct algorithm does not make a distinction between constructors, or other method types, and extracts a variable number of edges for `invokespecial`. It always produces one edge for the normal termination of the method, and either one or two edges for the exceptional flow of `NullPointerException` (`N.P.E.`), again depending on the presence of a suitable exception handler. Also, it produces one or two edges for each exception propagated by the invoked method n (denoted by $\mathcal{N}_m^{p,n}$). We present the proof for an arbitrary exception x, and generalize to all the possible propagated exceptions.

$$\mathcal{G}_{\text{JBC}}^{m,p,\texttt{invokespecial}} = \begin{cases} \circ_m^p \xrightarrow{n} \circ_m^{succ(p)}, \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\texttt{N.P.E.}}, \bullet_m^{p,\texttt{N.P.E.}} \xrightarrow{\varepsilon} \circ_m^q\} \cup \mathcal{N}_m^{p,n} & \text{if has handler} \\ \circ_m^p \xrightarrow{n} \circ_m^{succ(p)}, \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\texttt{N.P.E.},r}\} \cup \mathcal{N}_m^{p,n} & \text{otherwise} \end{cases}$$

$$\mathcal{N}_m^{p,n} = \begin{cases} \{\circ_m^p \xrightarrow{n} \bullet_m^{p,x}, \bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^t\} & \text{if has handler} \\ \{\circ_m^p \xrightarrow{n} \bullet_m^{p,x,r}\} & \text{otherwise} \end{cases}$$

Subcase I The receiver of an `invokespecial` is an object constructor. We instantiate the signature of the invoked method to n = `C` to stress this. It returns a sequence of assignments to temporary variables ($[\texttt{t}_1\texttt{:=l}_1\texttt{;t}_2\texttt{:=l}_2\texttt{;} \ldots]$), denoted by *HSave*; plus the call to `[new C]`.

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

Assignments to variables produce a single edge to the next control point. Thus, the extraction of *HSave* function produces a (path-like) graph corresponding to a weak transition $\circ_m^{\texttt{pc}} \Longrightarrow \circ_m^{\texttt{pc'}}$.

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

The rule for `[new C]` produces one normal edge for the case of successful execution, and one or two edges for to the exceptional flow of a `NullPointerException`. Also, it produces one or two edges for any exception x propagated from `C`.

verses nodes extracted from *HSave*), and $(\bullet_m^{p,\texttt{N.P.E.},r}, \bullet_m^{\texttt{pc'},\texttt{N.P.E.},r}) \in R$ since $seg_{JBC}(m,p) = seg_{BIR}(m,\texttt{pc'})$ and $\texttt{pc'} \in oap(s,\{\texttt{N.P.E.},r\})$.

The set of edges extracted for the propagation of an exception x also depends on the presence of a handler, and the explanation is similar to the one for `NullPointerException`. If there is a suitable handler, then there is an edge $\circ_m^p \xrightarrow{\texttt{C}} \bullet_m^{p,x}$, which is matched by $\circ_m^{\texttt{pc}} \xrightarrow{\texttt{C}}\!\!\!\!\Longrightarrow \bullet_m^{\texttt{pc'},x}$ (traversing the nodes from *HSave*), and $(\bullet_m^{p,x}, \bullet_m^{\texttt{pc'},x}) \in R$ since $seg_{JBC}(m,p) = seg_{BIR}(m,\texttt{pc'})$ and $\texttt{pc'} \in oap(s,\{x\})$. Also, there is the edge $\bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^t$, matched by $\bullet_m^{\texttt{pc'},x} \Longrightarrow \circ_m^{\texttt{pc}^t}$, and also $(\circ_m^t, \circ_m^{\texttt{pc}^t}) \in R$ since $seg_{JBC}(m,t) = seg_{BIR}(m,\texttt{pc}^t)$ and $\texttt{pc}^t = fst(seg_{BIR}(m,\texttt{pc}^t))$. If there is no handler, there is the edge $\circ_m^p \xrightarrow{\texttt{C}} \bullet_m^{p,x,r}$, which is matched by $\circ_m^{\texttt{pc}} \xrightarrow{\texttt{C}}\!\!\!\!\Longrightarrow \bullet_m^{\texttt{pc'},x,r}$. Thus $(\bullet_m^{p,x,r}, \bullet_m^{\texttt{pc'},x,r}) \in R$ since $seg_{JBC}(m,p) = seg_{BIR}(m,\texttt{pc'})$ and $\texttt{pc'} \in oap(s,\{x,r\})$.

Notice that if propagated exception $x = \texttt{N.P.E.}$, then `pc'` is the only possible control address tagging the node

$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc'},[\texttt{t}_\texttt{k}\texttt{:=new C}]} = \begin{cases} \{\circ_m^{\texttt{pc'}} \xrightarrow{\texttt{C}} \circ_m^{\texttt{pc'+1}}, \circ_m^{\texttt{pc'}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc'},\texttt{N.P.E.}}, \bullet_m^{\texttt{pc'},\texttt{N.P.E.}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}^q}\} \cup \bar{\mathcal{N}}_m^{\texttt{pc'},\texttt{C}} & \text{if has handler} \\ \{\circ_m^{\texttt{pc'}} \xrightarrow{\texttt{C}} \circ_m^{\texttt{pc'+1}}, \circ_m^{\texttt{pc'}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc'},\texttt{N.P.E.},r}\} \cup \bar{\mathcal{N}}_m^{\texttt{pc'},\texttt{C}} & \text{otherwise} \end{cases}$$

$$\bar{\mathcal{N}}_m^{\texttt{pc'},\texttt{C}} = \begin{cases} \{\circ_m^{\texttt{pc'}} \xrightarrow{\texttt{C}} \bullet_m^{\texttt{pc'},x}, \bullet_m^{\texttt{pc'},x} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}^t}\} & \text{if has handler} \\ \{\circ_m^{\texttt{pc'}} \xrightarrow{\texttt{C}} \bullet_m^{\texttt{pc'},x,r}\} & \text{otherwise} \end{cases}$$

Then $\star = \circ_m^{\texttt{pc}}$ since $\texttt{pc} = fst(s)$. The edge $\circ_m^p \xrightarrow{\texttt{C}} \circ_m^{succ(p)}$ is matched by $\circ_m^{\texttt{pc}} \xrightarrow{\texttt{C}}\!\!\!\!\Longrightarrow \circ_m^{\texttt{pc'+1}}$, which traverses all the nodes extracted from *HSave*, and also $(\circ_m^{succ(p)}, \circ_m^{\texttt{pc'+1}}) \in R$ since $seg_{JBC}(m,succ(p)) = seg_{BIR}(m,\texttt{pc'+1})$ and $\texttt{pc'+1} = fst(seg_{BIR}(m,\texttt{pc'+1}))$. That is, `pc'+1` is the first control address in the next code segment, after the current segment delimited by the `invokespecial` instruction.

The next set of edges depends on the presence of an exception handler for the `NullPointerException`. If there is a suitable handler, then there is another edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\texttt{N.P.E.}}$, which is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \bullet_m^{\texttt{pc'},\texttt{N.P.E.}}$ (also traversing nodes produced from *HSave*), and $(\bullet_m^{p,\texttt{N.P.E.}}, \bullet_m^{\texttt{pc'},\texttt{N.P.E.}}) \in R$ since $seg_{JBC}(m,succ(p)) = seg_{BIR}(m,\texttt{pc'})$ and $\texttt{pc'} \in oap(s,\{\texttt{N.P.E.}\})$. Also, there is the edge

with $XP = \{\texttt{N.P.E.}\}$, which has been shown above to be in R for either the cases of having or not a handler.

Subcase II The receiver of a call from `invokespecial` is a method within the same class, or from the super class. The $\texttt{BC2BIR}_{instr}$ transformation returns the `[notnull]` instruction, followed by a sequence of assignments (denoted by *HSave*), and the invocation instruction:

$\texttt{BC2BIR}_{instr}^{p,\texttt{invokespecial}}$
$= [\texttt{notnull};\mathit{HSave}(\texttt{pc},as);\texttt{t}_\texttt{k}\texttt{:=}e.n(\ldots)]$

Applying the extraction function to `[notnull]`, we have again a varying number of edges, depending on whether there is a handler for the `NullPointerException` or not:

$$\mathcal{G}_{\text{BIR}}^{m,\texttt{pc},[\texttt{notnull}]} = \begin{cases} \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc},\texttt{N.P.E.}}, \bullet_m^{\texttt{pc},\texttt{N.P.E.}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc}^q}\} & \text{if has handler} \\ \{\circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \circ_m^{\texttt{pc+1}}, \circ_m^{\texttt{pc}} \xrightarrow{\varepsilon} \bullet_m^{\texttt{pc},\texttt{N.P.E.},r},\} & \text{otherwise} \end{cases}$$

$\bullet_m^{p,\texttt{N.P.E.}} \xrightarrow{\varepsilon} \circ_m^q$, matched by $\bullet_m^{\texttt{pc'},\texttt{N.P.E.}} \Longrightarrow \circ_m^{\texttt{pc}^q}$, and again $(\circ_m^q, \circ_m^{\texttt{pc}^q}) \in R$ since $seg_{JBC}(m,q) = seg_{BIR}(m,\texttt{pc}^q)$ and $\texttt{pc}^q = fst(seg_{BIR}(m,\texttt{pc}^q))$.

If there is no handler, there is the edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\texttt{N.P.E.},r}$, which is matched by $\circ_m^{\texttt{pc}} \Longrightarrow \bullet_m^{\texttt{pc'},\texttt{N.P.E.},r}$ (and tra-

As explained above, the extraction of the assignments from the *HSave* function produces a (path-like) graph corresponding to a weak transition $\circ_m^{\texttt{pc+1}} \Longrightarrow \circ_m^{\texttt{pc'}}$.

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

The rule for [$\mathtt{t_k}$= e.n(...)] (i.e., *MethodCall*) produces one normal edge for the case of successful execution, and one or two edges for each propagated exception x.

$$\mathcal{G}_{\mathrm{BIR}}^{m,\mathtt{pc'},[\mathtt{t_k}:=e.n(\dots)]} = \{\circ_m^{\mathtt{pc'}} \xrightarrow{n} \circ_m^{\mathtt{pc'+1}}\} \cup \mathcal{N}_m^{\mathtt{pc'},n}$$

$$\mathcal{N}_m^{\mathtt{pc'},n} = \begin{cases} \{\circ_m^{\mathtt{pc'}} \xrightarrow{n} \bullet_m^{\mathtt{pc'},x}, \bullet_m^{\mathtt{pc'},x} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc}^t}\} & \text{if has handler} \\ \{\circ_m^{\mathtt{pc'}} \xrightarrow{n} \bullet_m^{\mathtt{pc'},x,r}\} & \text{otherwise} \end{cases}$$

Then $\star = \circ_m^{\mathtt{pc}}$ since $\mathtt{pc} = fst(s)$. The edge $\circ_m^p \xrightarrow{n} \circ_m^{succ(p)}$ is matched by $\circ_m^{\mathtt{pc}} \stackrel{n}{\Longrightarrow} \circ_m^{\mathtt{pc'+1}}$ which traverses $\circ_m^{\mathtt{pc+1}}$, and the nodes extracted from *HSave*. Thus $(\circ_m^{succ(p)}, \circ_m^{\mathtt{pc'+1}}) \in R$ since $seg_{JBC}(m, succ(p)) = seg_{BIR}(m, \mathtt{pc'+1})$ and $\mathtt{pc'+1} = fst(seg_{BIR}(m, \mathtt{pc'+1}))$.

The set of edges extracted because of a potential NullPointerException depends on the presence of a handler. If there is a suitable handler, then there is an edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\mathtt{N.P.E.}}$, which is matched by $\circ_m^{\mathtt{pc}} \Longrightarrow \bullet_m^{\mathtt{pc,N.P.E.}}$, and $(\bullet_m^{p,\mathtt{N.P.E.}}, \bullet_m^{\mathtt{pc,N.P.E.}}) \in R$ since $seg_{JBC}(m, p) = seg_{BIR}(m, \mathtt{pc})$ and $\mathtt{pc} \in oap(s, \{\mathtt{N.P.E.}\})$. Also, there is an edge $\bullet_m^{p,\mathtt{N.P.E.}} \xrightarrow{\varepsilon} \circ_m^q$, which is matched by $\bullet_m^{\mathtt{pc,N.P.E.}} \Longrightarrow \circ_m^{\mathtt{pc}^q}$ and $(\circ_m^q, \circ_m^{\mathtt{pc}^q}) \in R$ since $seg_{JBC}(m, q) = seg_{BIR}(m, \mathtt{pc}^q)$ and $\mathtt{pc}^q = fst(seg_{BIR}(m, \mathtt{pc}^q))$. If there is no handler, then there is an edge $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,\mathtt{N.P.E.},r}$, which is matched by $\circ_m^{\mathtt{pc}} \Longrightarrow \bullet_m^{\mathtt{pc,N.P.E.},r}$ and $(\bullet_m^{p,\mathtt{N.P.E.},r}, \bullet_m^{\mathtt{pc,N.P.E.},r}) \in R$ since $seg_{JBC}(m, p) = seg_{BIR}(m, \mathtt{pc})$ and $\mathtt{pc} \in oap(s, \{\mathtt{N.P.E.}, r\})$. Notice that in either of the cases pc may not be the only control address tagging a node with N.P.E., and $\bullet_m^{p,\mathtt{N.P.E.}}$ or $\bullet_m^{p,\mathtt{N.P.E.},x}$ must also relate to such node.

Again, the explanation for edges related to a propagated exception x is similar to the one for N.P.E.. If there is a handler for x, then there is an edge $\circ_m^p \xrightarrow{n} \bullet_m^{p,x}$, which is matched by $\circ_m^{\mathtt{pc}} \stackrel{n}{\Longrightarrow} \bullet_m^{\mathtt{pc'},x}$, which traverses $\circ_m^{\mathtt{pc+1}}$ and the nodes extracted from *HSave*, and $(\bullet_m^{p,x}, \bullet_m^{\mathtt{pc'},x}) \in R$ since $\mathtt{pc'} \in oap(s, \{x\})$. There is also an edge $\bullet_m^{p,x} \xrightarrow{\varepsilon} \circ_m^t$ which is matched by $\bullet_m^{\mathtt{pc'},x} \Longrightarrow \circ_m^{\mathtt{pc}^t}$, and also $(\circ_m^t, \circ_m^{\mathtt{pc}^t}) \in R$ since $seg_{JBC}(m, t) = seg_{BIR}(m, \mathtt{pc}^t)$ and $\mathtt{pc}^t = fst(seg_{BIR}(m, \mathtt{pc}^t))$. If there is no handler, then there is an edge $\circ_m^p \xrightarrow{n} \bullet_m^{p,x,r}$, that is matched by $\circ_m^{\mathtt{pc}} \stackrel{n}{\Longrightarrow} \bullet_m^{\mathtt{pc'},x,r}$ (traversing $\circ_m^{\mathtt{pc+1}}$, the nodes from *HSave* and $\circ_m^{\mathtt{pc'}}$), and $(\bullet_m^{p,x,r}, \bullet_m^{\mathtt{pc'},x,r}) \in R$ since $seg_{JBC}(m, p) = seg_{BIR}(m, \mathtt{pc'})$ and $\mathtt{pc'} \in oap(s, \{x, r\})$.

Notice that if the propagated exception $x = \mathtt{N.P.E.}$, then both nodes $\bullet_m^{\mathtt{pc},x,r}$ and $\bullet_m^{\mathtt{pc'},x,r}$ relate to $\bullet_m^{p,x,r}$ in R. The same is true for the case where there is a handler, and the corresponding exceptional nodes $\bullet_m^{\mathtt{pc},x}$ and $\bullet_m^{\mathtt{pc'},x}$ relate to $\bullet_m^{p,x}$ in R. This concludes the case, and the whole case analysis.

Let now $(\circ_m^{p,r}, \star) \in R_2$. The proof proceeds with the RETINST set, the only type of the producer instructions of the bytecode segment $seg_{JBC}(m, p)$ giving rise to $\circ_m^{p,r}$. All return instructions are producer instructions. However, the direct algorithm does not produce edges for them, but simply adds the atomic proposition r to the normal sink nodes tagged with the address of the return instruction. Let p be the control address of the return instruction. The transformation $\mathtt{BC2BIR}_{instr}$ returns a single instruction, applied to which $\mathcal{G}_{\mathrm{BIR}}$ produces a single edge:

$$\mathtt{BC2BIR}_{instr}^{p,\mathtt{return}} = [\mathtt{return}\ expr()]$$
$$\mathcal{G}_{\mathrm{BIR}}^{m,\mathtt{pc},[\mathtt{return}\ expr]} = \{\circ_m^{\mathtt{pc}} \xrightarrow{\varepsilon} \circ_m^{\mathtt{pc},r}\}$$

In this case we have to relate $\circ_m^{p,r}$ via R_2 rather than via R_1. Then $\star = \circ_m^{\mathtt{pc},r}$ since $seg_{JBC}(p) = seg_{BIR}(m, \mathtt{pc})$ and $\mathtt{pc} \in oap(s, \{r\})$. That is, pc tags the only node where the set $XP = \{r\}$. Since there is no outgoing edge from $\circ_m^{p,r}$, this concludes the case of return instructions and the whole proof. □

References

1. Allen, F.E.: Control flow analysis. SIGPLAN Not. **5**, 1–19 (1970). doi:10.1145/390013.808479
2. Amighi, A.: Flow graph extraction for modular verification of java programs. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden (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., Gomes, PdC, Gurov, D., Huisman, M.: Sound control-flow graph extraction for Java programs with exceptions. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Software Engineering and Formal Methods, Lecture Notes in Computer Science, vol. 7504, pp. 33–47. Springer, Berlin (2012). doi:10.1007/978-3-642-33826-7_3
4. Armando, A., Costa, G., Merlo, A., Verderame, L.: Enabling byod through secure meta-market. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & #38; Mobile Networks, WiSec '14, pp. 219–230. ACM, New York (2014). doi:10.1145/2627393.2627410. http://doi.acm.org/
5. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA, pp. 324–341 (1996)
6. Barre, N., Demange, D., Hubert, L., Monfort, V., Pichardie, D.: SAWJA API documentation (2011). http://javalib.gforge.inria.fr/doc/sawja-api/sawja-1.3-doc/api/index.html
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 '99, 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. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Z.H.: Bandera: extracting finite-state models from java source code. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE '00, pp. 439–448. ACM, New York (2000). doi:10.1145/337180.337234. http://doi.acm.org/
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the 9th European Conference on Object-Oriented Programming, ECOOP, pp. 77–101. Springer, London (1995). http://dl.acm.org/citation.cfm?id=646153.679523
11. 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
12. Dwyer, M.B., Hatcliff, J., Joehanes, R., Laubach, S., Păsăreanu, C.S., Zheng, H., Visser, W.: Tool-supported program abstraction for finite-state verification. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, pp. 177–187. IEEE Computer Society, Washington, DC (2001). http://dl.acm.org/citation.cfm?id=381473.381493
13. Freund, S.N., Mitchell, J.C.: A type system for the Java bytecode language and verifier. J. Autom. Reason. **30**, 271–321 (2003)
14. Gomes, P.D.C.: Sound modular extraction of control flow graphs from java bytecode. Licentiate Thesis, KTH Royal Institute of Technology (2012). http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-105275 QC 20121122
15. Gomes, P.D.C., Picoco, A., Amighi, A.: ConFlEx (2012). http://www.csc.kth.se/pedrodcg/conflex
16. Gomes, PdC, Picoco, A., Gurov, D.: Sound control flow graph extraction from incomplete java bytecode programs. In: Gnesi, S., Rensink, A. (eds.) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 8411, pp. 215–229. Springer, Berlin (2014). doi:10.1007/978-3-642-54804-8_15
17. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Cavalli, A.: Formal characterization of illegal control flow in android system. In: 2013 International Conference on Signal-Image Technology Internet-Based Systems (SITIS), pp. 293–300 (2013). doi:10.1109/SITIS.2013.56
18. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Inf. Comput. **206**(7), 840–868 (2008)
19. 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, pp. 92–106. Springer, Berlin (2010)
20. 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, Berlin (2008)
21. 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, Berlin (2010)
22. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. SIGPLAN Not. **42**, 3–8 (2007)
23. 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)
24. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped—a model-checker for pushdown systems (2005). http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/
25. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). doi:10.1145/360248.360252
26. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
27. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification. java se 7 edition. Tech. Rep. JSR-000924, Oracle (2012)
28. Mihancea, P., Minea, M.: Jmodex: Model extraction for verifying security properties of web applications. In: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), pp. 450–453 (2014). doi:10.1109/CSMR-WCRE.2014.6747216
29. Milner, R.: Communicating and Mobile Systems: the π-Calculus, chap. 6. Cambridge University Press, New York (1999)
30. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE, New York (1977). doi:10.1109/SFCS.1977.32
31. Schwoon, S.: Model-checking pushdown systems. Ph.D. thesis, Technische Universität München (2002)
32. 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 '99, pp. 265–276. IEEE Computer Society, New York (1999)
33. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. **26**, 849–871 (2000). doi:10.1109/32.877846
34. Soleimanifard, S., Gurov, D.: Algorithmic verification of procedural programs in the presence of code variability. In: Post-Proceedings of the 11th International Symposium on Formal Aspects of Component Software (FACS'14), Lecture Notes in Computer Science, vol. 8997. Springer, Berlin (2014)
35. Soleimanifard, S., Gurov, D., Huisman, M.: ProMoVer Web Interface (2012). http://www.csc.kth.se/siavashs/ProMoVer
36. Soleimanifard, S., Gurov, D., Huisman, M.: Procedure-modular specification and verification of temporal safety properties. Software & Systems Modeling, pp. 1–18 (2013). doi:10.1007/s10270-013-0321-0. http://dx.doi.org/
37. Spoto, F.: Precise null-pointer analysis. Softw. Syst. Model. **10**(2), 219–252 (2011). doi:10.1007/s10270-009-0132-5
38. 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 '00, pp. 264–280. ACM, New York (2000). doi:10.1145/353171.353189. http://doi.acm.org/
39. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E. Co, P.: Soot—A Java Optimization Framework. In: CASCON '99, pp. 125–135 (1999). http://www.sable.mcgill.ca/soot/
40. Watson, T.J.: IBM: Libraries for Analysis (Wala) (2012). http://wala.sourceforge.net/
41. Zhao, J.: Analyzing control flow in Java bytecode. In: Proceedings of the 16th Conference of Japan Society for Software Science and Technology, pp. 313–316 (1999)