Sound Control Flow Graph Extraction from Incomplete Java Bytecode Programs

Pedro de Carvalho Gomes, Attilio Picoco, and Dilian Gurov

KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. The modular analysis of control flow of incomplete Java bytecode programs is challenging, mainly because of the complex semantics of the language, and the unknown inter-dependencies between the available and unavailable components. In this paper we describe a technique for incremental, modular extraction of control flow graphs that are provably sound w.r.t. sequences of method invocations and exceptions. The extracted models are suitable for various program analyses, in particular model-checking of temporal control flow safety properties. Soundness comes at the price of over-approximation, potentially giving rise to false positives reports during verification. Still, our technique supports incremental refinement of the already extracted models, as more components code becomes available. The extraction has been implemented as the ConFlEx tool, and test-cases show its utility and efficiency.

1 Introduction

The main obstacle to the formal verification of software is the size of its state space. A standard approach to address this problem is to construct an abstract model of manageable size and to perform the verification over the model. Ideally, the abstraction should come with a formal argument that it is property-preserving for the class of properties of interest, otherwise the verification results cannot be trusted. *Control flow graphs* (CFGs) are among the most commonly used software models, where nodes represent the program's control points, while edges represent the transfer of control between the points.

In this paper we present a framework for the extraction of CFGs from the available components of *incomplete Java bytecode* (JBC) *programs.* That is, programs where the implementation of some components is not yet available. Typical situations when one has to deal with incomplete programs are systems under development, or systems depending on third-party software. In the latter case, it is common that the source code of the third-party software never becomes available, which motivates our choice to analyze Java bytecode.

We extract CFGs that are *sound* w.r.t sequences of method invocations and exceptions. Such models are useful for many static analyses, especially for the formal verification of temporal control flow safety properties. Previous techniques have been proposed to analyze incomplete JBC programs [6,16]; however they are admittedly unsound. To the best of our knowledge, our framework is the first to soundly analyze the control flow of incomplete programs.

The challenges to soundly analyze control flow from incomplete JBC programs are twofold. The first are the *object-oriented* features of JBC. For instance, *virtual method calls* (VMC) and exceptions impose difficulties. The second are the unknown inter-dependencies between available and yet unavailable software components. For instance, it is hard to estimate the control flow caused by exception propagation, or to determine precisely the possible receivers of a VMC.

We define our framework by generalizing a previous algorithm from Amighi *et al.* [2] for complete JBC programs that uses a transformation into an intermediate bytecode representation (BIR) [12]. The transformation into BIR allows the *precise* estimation of a significant subset of the implicit (e.g., division by zero) exceptions, and of explicit (with `athrow` instruction) exceptions.

The inter-dependencies involving yet unavailable components are captured by means of *user-provided interfaces*. Our approach is conservative, and assumes that unavailable methods may propagate any exception. This results in significant over-approximation, but the user may alleviate it by specifying in the method's interface the exceptions it should never propagate.

Still, valid global properties may fail to be established, giving rise to so-called *false positives*. The algorithm mitigates this by allowing the *incremental refinement* of previously extracted CFGs, as more code becomes available. This is accomplished by decoupling the intra- and inter-procedural exceptional flow analysis. So, properties that could not be verified in the more abstract CFGs may be established over the refined CFGs.

The framework defines formally the constraints to instantiating yet unavailable code, needed to ensure the soundness of the already generated CFGs w.r.t. sequences of method invocations and exceptions. Further, we prove the *correctness* of our extraction. First, we show that the extracted CFGs from the available components are supergraphs of the ones extracted from the same components by the algorithm for complete programs. Then, we connect this with previously established results to conclude that the CFGs extracted with the present algorithm are also sound w.r.t. the JBC behavior (as defined by the JVM), as long as the specified constraints are respected. Therefore, already established behavioral or structural properties are thus guaranteed to still hold.

We have implemented our technique as the ConFlEx tool. It features caching of previous analyses, necessary for the incremental refinement, and matching of newly arriving code against their interface specifications. Our experimental results confirm the intuitive expectation that the over-approximations impact significantly the size of the CFGs. Also, the results show that ConFlEx is efficient, and performs a light-weight extraction of CFGs.

Organization. Section 2 describes the program models on which we base our technique, and the transformation into the BIR. Section 3 motivates our work by presenting a compositional verification technique that benefits directly from our results. Section 4 describes our framework to analyze incomplete programs, and outlines a correctness argument. Section 5 describes the implementation of our approach, and presents experimental results. Section 6 discusses related work, while Section 7 draws conclusions and outlines directions for future work.

2 Preliminaries

In this section we briefly present the program model, and give an overview of the BIR language, both necessary to define our CFG extraction algorithm.

2.1 Program Model

We define CFGs, following Huisman *et al.* [13], as *Kripke structures* with transition labels, where nodes represent program control points, and the edges represent how instructions shift control between the points. The atomic propositions associated with nodes contain information about the control address, possible exceptions, and returns. We use the following notational convention: $\circ_m^p$ denotes a normal non-return control node in the address p of method m, $\bullet_m^{p;x}$ an exceptional non-return control node with exception x, while $\circ_m^{p;r}$ and $\bullet_m^{p;x,r}$ a normal and an exceptional return node, respectively.

Edge labels are either method signatures m corresponding to invocation instructions, or the special label ε signifying any other type of instruction. This choice is made here because of our interest in the possible sequences of method invocations (expressed as temporal safety properties), but the program model can be adapted to other needs as well. API methods are not considered a part of the program, and are thus labeled by ε. However, the propagated exceptions declared in the signature with `throws` are taken into account.

Let Meth and Excp be the sets of all method signatures and exceptions, respectively. We now define formally CFGs as a collection of method graphs.

Definition 1 (Method Graph). *A* method graph *for method $m \in M$ over sets $M \subseteq$ Meth and $E \subseteq$ Excp is a pair $\mathcal{G}_m = (\mathcal{M}_m, \mathbb{E}_m)$, where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a labeled Kripke structure, with V_m the set of control nodes of m, $A_m = \{m, r\} \cup E$ the set of atomic propositions, and $L_m = M \cup \{\varepsilon\}$ the set of transition labels. We require that $m \in \lambda_m(v)$ 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., every control node is tagged with the method signature it belongs to and with at most one exception). $\mathbb{E}_m \subseteq V_M$ is the (non-empty) set of entry control points of m.*

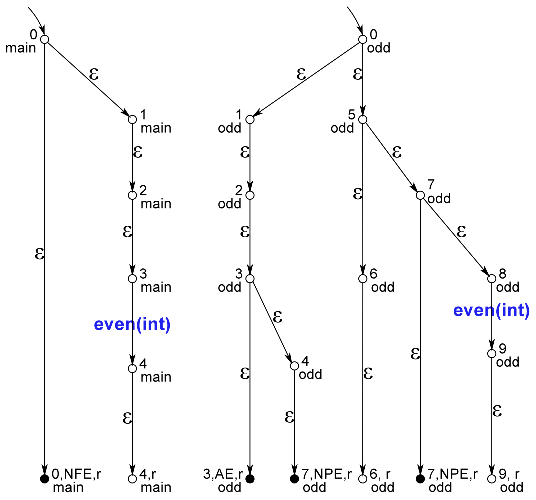
Every control flow graph $\mathcal{G}$ is equipped with an *interface* $I = (I^+, I^-, I^e)$, written $\mathcal{G} : I$, specifying the (disjoint) sets of *provided* and (externally) *required* methods, and the set $I^e \subseteq I^+ \times E$ of potentially propagated exceptions by the provided methods. We say a CFG is *closed* if there are no (externally) required methods; we say it is *open* otherwise. CFG *composition* is defined as the disjoint union $\uplus$ of their method graphs. Interface composition is defined as $I_1 \cup I_2 = (I_1^+ \cup I_2^+, (I_1^- \cup I_2^-) \backslash (I_1^+ \cup I_2^+), I_1^e \cup I_2^e)$.

Example 1 (CFG). Figure 1a shows a simple program to check the parity of an integer. It is presented in Java source (rather than bytecode), to help the comprehension. The program has three methods. The method `main` calls `parseInt` to convert the input string into an integer, then calls `even`. Notice that `parseInt` is a method from the Java API, and is not considered a part of the program. However, its signature declares that it may propagate a `NumberFormatException`, and this must be taken into account in the analysis. The method `odd` potentially throws an `ArithmeticException`.

The implementation of method `even` is not available. We specify it with the interface I_{even} = ({`even`}, {`odd`}, {}). It declares that the method may call itself or `odd`, and does not propagate any exceptions. It is represented in the code by the empty-bodied method, and the Java annotation `GhostComponent`.

```
public class EvenOdd{

public static void main(String[] argv){
   EvenOdd obj = new EvenOdd();
   obj.even(Integer.parseInt(argv[0]));
}

public boolean odd(int n){
   if (n < 0)
      throw new ArithmeticException();
   else if (n == 0)
      return false;
   else
      return even(n-1);
}

/*** Unavailable method ***/
@GhostComponent( handlers={"any"},
req_meths={"odd(int)"} )
public boolean even(int n) {};

}
```

(a) Program source



(b) CFGs for available methods

Fig. 1. Example of Incomplete Java program

Figure 1b shows the CFGs for the available methods `main` and `odd`. The nodes are tagged with the method's signature and a control address. Entry nodes are depicted as usual by incoming edges without source. There are three exceptional nodes in the CFG, which represent points in which program control is taken over by the JVM to take care of the exception. These three are also return nodes (i.e., tagged with the atomic proposition r), and indicate the propagation of the respective exception by the method. The invocations of methods `even` and `odd` are represented by call edges. The invocation of `parseInt`, however, which is a method from the Java API, is not represented by a call edge. Further, the method's signature declares that a `NumberFormatException` (`NFE`) is potentially propagated, and this is reflected by an edge to $\bullet^{0,\mathtt{NFE},r}_{\mathtt{main}}$.

2.2 Bytecode Intermediate Representation

The BIR language is an intermediate representation of Java bytecode developed at INRIA Rennes [12]. The main difference with JBC is that BIR instructions are stack-less. That is, instructions do not operate over values stored on the operand stack. Instead, a JBC method is translated into BIR by symbolically executing the bytecode, using an abstract stack. This stack is used to reconstruct expression trees and to connect instructions to its operands. We give a brief overview of the BIR language. However, we omit the details of the transformation from JBC to BIR; for a full account we refer to [12]. Figure 2a shows the BIR syntax.

```
      lvar ::= x | x1 | ...     tvar ::= t | t1 | ...
               this             expr ::= c | null
    target ::= lvar                    | expr ⊕ expr
             | tvar                    | tvar | lvar
             | expr.f                  | expr.f

 Assignment ::= target := expr
     Return ::= return expr | return
 MethodCall ::= expr.ns(expr, ...)
              | target := expr.ns(expr, ...)
  NewObject ::= target := new C(expr, ...)
  Assertion ::= notnull expr | notnegsize expr
              | notzero expr | checkbound expr
Instruction ::= nop | if expr pc | goto pc
              | throw expr | mayinit C
              | Assignment | Return
              | MethodCall | NewObject
              | Assertion
```

(a) Syntax

public boolean odd(int x)

Java bytecode	BIR
0: iload x	
1: ifge 12	0: if (x >= 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 x	
13: ifne 18	5: if (x != 0) goto 7
16: iconst 0	
17: ireturn	6: return 0
18: aload 0	
19: iload x	
20: iconst 1	
21: isub	7: notnull this
22: invokevirtual even(int)	8: t0 := this.even(x - 1)
25: ireturn	9: return t0

(b) Comparison with JBC

Fig. 2. The BIR language

The transformation into BIR simplifies the analysis of exceptional control flow. It identifies *implicit* exceptions by inserting special assertions before the instructions that can potentially raise the exception, as defined by the JVM specification [18]. For example, the transformation inserts a [notnegsize *expr*] assertion before instructions that might raise a NegativeArraySizeException. If the assertion holds, meaning that *expr* does not evaluate to negative a number, 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. Moreover, the BIR transformation connects the *explicit* exceptions, raised by athrow, to their types in the [throw *target*] instruction. Now, data-flow analysis can estimate the possible types of the *target* variable.

Example 2 (JBC and BIR Comparison). Figure 2b shows the JBC and BIR versions of method odd() from Figure 1a. The different shades indicate the reconstruction of expression trees, and the collapsing of instructions by the transformation. The BIR method has a local variable (x), which is also present in the JBC, and a newly introduced variable (t0). Notice that the argument for the method invocation and the operand to the [if] instruction are reconstructed expression trees. The [notnull] instruction asserts that NullPointerException can potentially be raised at this program point.

3 Motivation

The motivation for the present work is to support the formal verification of incomplete JBC programs. Typical scenarios of incomplete programs are systems under development, or systems that depend on third-party software. Two examples are an ATM system that depends on the code from users' smart-cards, or ERP systems, which are typically modular. It is desirable that the available components are checked against global properties in advance. Then, the only pending task is the verification of the missing code, which should be light-weight, and can be delayed until the user inserts the smart-card into the ATM, or a module of the ERP system is provided.

One technique that enables the verification of incomplete programs is the compositional principle developed by Gurov *et al.* [11]. There, unavailable software components are represented with an interface and a local temporal specification. Both are used to compute a so-called *maximal model*, i.e., a model that simulates the behavior of any model that respects the interface and satisfies the local specification, and can thus represent the unavailable component when checking global temporal safety properties. Once the missing code becomes available, it is checked to match the interface and the local specification. If it does, this entails the global properties.

The correctness of the verified temporal safety properties is only guaranteed for models that *soundly* over-approximate the actual program behavior. Soundness, however, comes at the price of excessive over-approximations. Thus, potentially giving rise to false positives. To alleviate this problem, we aim to a model extraction strategy that is *incremental*: whenever more code arrives, the existing model can be refined, and the false positive may now be provable.

Example 3 (Compositional Verification). Suppose we want to verify two global properties over the available code from the incomplete program in Figure 1. Let ϕ_1 be defined informally as "if an `ArithmeticException` is raised within a method, it must be either caught locally, or by the immediate caller method", and ϕ_2 be the same property, but for an `ArrayStoreException`.

We define the local property $\psi_{\texttt{even}}$ for the missing method `even` informally as "after calling `odd`, `even` must terminate normally", and construct the maximal CFG for $\psi_{\texttt{even}}$ and $I_{\texttt{even}}$. Also, we extract the CFGs from the available methods `main` and `odd`, and compose them with the maximal CFG for `even`.

The global property ϕ_1 is checked against the composed model, and it turns out to hold. Thus, once the implementation of `even` is provided, we simply extract its CFG, and check it against the local property $\psi_{\texttt{even}}$. If it holds, the correctness of the program is established w.r.t. ϕ_1. Also the property ϕ_2 is checked over the same composed model. However, ϕ_2 does not hold since neither the interface, nor the local property restrict an `ArrayStoreException` from being raised by `even`. Still, it may be a false positive: once the code of `even` becomes available, we may extract its CFG, refine the previously extracted CFGs, compose them, and re-check the property.

4 CFG Extraction Framework

In this section we outline the theoretical definitions of the framework for extraction of CFGs from incomplete JBC programs, and summarize the soundness argument. For the complete definitions and results we refer to [4].

4.1 Incomplete JBC Programs and Extraction Algorithm

We model incomplete JBC programs as *open environments*, following Freund and Mitchell's definition of *closed environment* for complete JBC programs [10]. An open environment Γ_o is defined in Figure 3 as the union of partial mappings from method references, class names and `interface` names to their respective definitions. We write `interfaces` (in typewrite font) to distinguish it from the CFG interfaces introduced in Section 2.1. An important aspect of the definition is that it contains all information about the type hierarchy. Thus, we can enumerate the set of exception types from a given open environment.

The difference from the modeling of complete programs is that in open environments a method body (i.e., `code`) may be empty. Also, entries in the `handlers` have a special meaning for empty methods. They represent the exception types that cannot be propagated by the method's implementation, once provided.

$$
\begin{array}{ll}
\Gamma^I : \textit{IFace-Name} \rightharpoonup \left\langle \begin{array}{rl} \texttt{interfaces} : & \textit{set of IFace-Name} \\ \texttt{method} : & \textit{set of IFace-Method-Ref} \end{array} \right\rangle & \\
\Gamma^C : \textit{Class-Name} \rightharpoonup \left\langle \begin{array}{rl} \texttt{super} : & \textit{Class-Name} \\ \texttt{interfaces} : & \textit{set of IFace-Name} \\ \texttt{fields} : & \textit{set of Field-Ref} \end{array} \right\rangle & \Gamma_o = \Gamma^I \cup \Gamma^C \cup \Gamma^M \\
\Gamma^M : \textit{Method-Ref} \rightharpoonup \left\langle \begin{array}{rl} \texttt{code} : & \textit{Instruction}^* \\ \texttt{handlers} : & \textit{Handler}^* \end{array} \right\rangle &
\end{array}
$$

Fig. 3. Open environment of a JBC/BIR program

Open environments also model the BIR version of incomplete programs. The differences to the JBC version is the `code` array, translated syntactically to BIR instructions, and `handlers`, which has the addresses of the exception handlers mapped to the respective BIR addresses. We use the common modeling as open environments to define the CFG extraction indirectly. First we transform JBC into BIR; then extract CFGs from the intermediate representation. Here we focus on the latter transformation; for the former, we refer again to [12].

The $\mathtt{o}\mathcal{G}$ algorithm extracts CFGs from the available methods of an open environment. It iterates over the instructions array of a method m and produces, for every program counter `pc` and corresponding instruction i, a set of edges $\mathtt{o}\mathcal{G}_m^{\mathtt{pc},i}$ together with the associated nodes. Figure 4 shows the necessary auxiliary functions, and the extraction rules for $\mathtt{o}\mathcal{G}$, grouped by their BIR instruction type.

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

$$\mathcal{H}_m^{\mathtt{pc},x,l} = \begin{cases} \{\ (\circ_m^{\mathtt{pc},x}, l, \bullet_m^{\mathtt{pc},x,r})\ \} & \text{if } h_m^{\mathtt{pc},x} = \mathit{undef} \\ \{\ (\circ_m^{\mathtt{pc'}}, l, \bullet_m^{\mathtt{pc},x}), (\bullet_m^{\mathtt{pc},x}, \varepsilon, \circ_m^{\mathtt{pc'}})\ \} & \text{if } h_m^{\mathtt{pc},x} = \mathtt{pc'} \end{cases}$$

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

$$\mathtt{o}\mathcal{G}_m^{\mathtt{pc},i} = \begin{cases} \{(\circ_m^{\mathtt{pc}}, \varepsilon, \circ_m^{\mathtt{pc+1}})\} & \text{if } i \in \mathit{Assignment}\ \cup\ \{\mathtt{[nop]},\mathtt{[mayinit]}\} \\ \{(\circ_m^{\mathtt{pc}}, \varepsilon, \circ_m^{\mathtt{pc+1}}), (\circ_m^{\mathtt{pc}}, \varepsilon, \circ_m^{\mathtt{pc'}})\} & \text{if } i = \mathtt{[if}\ \mathit{expr}\ \mathtt{pc']} \\ \{(\circ_m^{\mathtt{pc}}, \varepsilon, \circ_m^{\mathtt{pc'}})\} & \text{if } i = \mathtt{[goto\ pc']} \\ \{(\circ_m^{\mathtt{pc}}, \varepsilon, \circ_m^{\mathtt{pc},r})\} & \text{if } i \in \mathit{Return} \\ \bigcup_{\{x \mid x <: X\}} \mathcal{H}_m^{\mathtt{pc},x,\varepsilon} & \text{if } i = \mathtt{[throw}\ X\mathtt{]} \\ \{(\circ_m^{\mathtt{pc}}, \varepsilon, \circ_m^{\mathtt{pc+1}})\} \cup \mathcal{H}_m^{\mathtt{pc},\chi_i,\varepsilon} & \text{if } i \in \mathit{Assertion} \\ \{(\circ_m^{\mathtt{pc}}, \mathtt{C}, \circ_m^{\mathtt{pc+1}})\} \cup \mathcal{H}_m^{\mathtt{pc},\mathtt{NPE},\mathtt{C}} \cup \mathcal{N}_m^{\mathtt{pc},\mathtt{C}} & \text{if } i \in \mathit{NewObject} \\ \bigcup_{n \in MCA(\mathtt{n})} \{(\circ_m^{\mathtt{pc}}, n, \circ_m^{\mathtt{pc+1}})\} \cup \mathcal{N}_m^{\mathtt{pc},n} & \text{if } i \in \mathit{MethodCall} \end{cases}$$

Fig. 4. CFG extraction from incomplete BIR

Let $h_m^{\mathtt{pc},x}$ denote the first handler (if any) in the exception table of method m (with the same entries as the JBC table, but with control points relating to BIR instructions) for the exception of type (or subtype of) x at position pc. The function $\mathcal{H}_m^{\mathtt{pc},x,l}$ produces edges related to exception handling, determined by the value of $h_m^{\mathtt{pc},x}$. If there is a handler for x at pc in m, it returns two edges: one from a normal node to an exceptional node, and another one from the exceptional node to the normal node tagged with the handler's initial control point pc'; otherwise, it returns an edge to an exceptional return node. The label l is either the signature of a callee method that propagates the exception, or ε, if the exception is raised within the method. The function χ_i simply returns the exception type associated to a BIR assertion i.

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

The $\mathtt{o}\mathcal{G}$ algorithm is a generalization from the $\mathcal{G}$ algorithm, proposed by Amighi *et al.* [2] for complete programs. It introduces two significant modifications. The first one is w.r.t. virtual method call resolution. The $\mathcal{G}$ algorithm is parametrized by a sound VMC resolution algorithm. However, standard VMC algorithms, such as the *Rapid Type Analysis* (*RTA*) [3], are defined for complete programs only, and may provide unsound estimation in the absence of code. We therefore fix the VMC resolution algorithm to our *Modular Class Analysis* (*MCA*), which is a generalization of the *Class Hierarchy Analysis* (*CHA*) [7]. *MCA* soundly over-approximates the possible receivers to methods with the same signature (`ns`) from sub-types and from the closest super-type of the static type (`C`) that are either provided or declared to be missing (given by function *dom*). The second modification concerns the function $\mathcal{N}$ that computes the control flow caused by exception propagation. In this case, when the callee method is unavailable, the set of exceptions that are propagated is defined as all exception types, excluding those annotated by the user in `handlers` to be never propagated.

4.2 Correctness of $\circ\mathcal{G}$

The main purpose of the transformation above is to extract CFGs from the available components of incomplete JBC programs that are sound for any instantiation of the missing code. CFGs that preserve this property entails the verification of global temporal safety properties, as explained in Section 3. Further, the transformation allows the extracted CFGs to be refined incrementally as more component code becomes available, until completion of the system.

Theoretically, both purposes are supported through a *refinement pre-order* on open environments, as defined below. Notice that closed environments for complete programs are simply open environments where all method bodies are provided, and are thus minimal w.r.t. the pre-order.

Definition 2 (Environment Refinement). *Let Γ_o and Γ_o' be open environments. We say that environment Γ_o* refines *environment Γ_o', written $\Gamma_o \preceq \Gamma_o'$, if the following conditions hold:*

(*i*) *method references, class names and* `interface` *names defined in Γ_o' must also be in Γ_o;*
(*ii*) *an* `interface` *in Γ^I contain the same methods, and extend a subset of the* `interfaces` *in $\Gamma^{I'}$;*
(*iii*) *classes in Γ^C have the same super-class, implement a subset of the* `interfaces` *in of the same classes in $\Gamma^{C'}$;*
(*iv*) *a method in Γ_o' must have a superset of the handlers of Γ_o' if it is unavailable in both environments, it must have the same* `code` *and* `handlers` *if it is implemented in both environments, or the method implementation Γ_o cannot propagate exceptions declared in Γ_o'.*`handlers`*, where it was unavailable.*

We say that Γ implements *Γ_o whenever $\Gamma \preceq \Gamma_o$ and Γ is closed.*

The refinement of a method which is unavailable in both environments entails that in Γ_o it propagates at most the same set of exceptions as in Γ_o'. Thus, a CFG extraction from Γ_o' must have over-approximated the set of propagated exceptions involving the method. In the refinement which a method is implemented in both environments, there cannot be changes; otherwise, the method graph extracted from Γ_o' would not soundly over-approximate the method graph from Γ_o. The refinement of a missing method in Γ_o', which is implemented in Γ_o, simply guarantees that it respects its interface w.r.t. propagated exceptions.

The following result states that, when applied to closed environments, the algorithm for open environments reduces to the one for closed environments with *MCA* as the virtual method call resolution algorithm.

Theorem 1. *Let Γ be a closed environment, and $\mathcal{G}_{MCA}$ be the instantiation of $\mathcal{G}$ with MCA. Then $\mathcal{G}_{MCA}(\Gamma) = \mathsf{o}\mathcal{G}(\Gamma)$.*

The next result establishes *monotonicity* of CFG extraction w.r.t. refinement.

Theorem 2. *Let Γ_o and Γ_o' be open environments, and m be the signature of a method available on both. Then $\Gamma_o \preceq \Gamma_o'$ implies $\mathsf{o}\mathcal{G}(m, \Gamma_o) \subseteq \mathsf{o}\mathcal{G}(m, \Gamma_o')$.*

The proofs of the above theorems are available in [4], due to space limitation. These results ensure *soundness* of the CFG extraction w.r.t. temporal safety properties, by virtue of several results established earlier. Here we briefly outline the soundness argument; for the full account the reader is referred to [2,11]. First, subgraph inclusion of CFGs entails *structural simulation* between CFGs in terms of a simulation relation between the nodes of the two graphs. Next, structural simulation in turn entails *behavioral simulation* in terms of a simulation relation between the behavioral configurations induced by the two graphs by means of pushdown systems ([11, Th. 36]). Third, temporal safety properties are preserved (backwards) under behavioral simulation ([11, Cor. 17]). These three results guarantee preservation of temporal safety properties under refinement of open environments. Together with the soundness result for $\mathcal{G}$ established in [2] and Theorem 1 above, we obtain soundness of $\mathsf{o}\mathcal{G}$.

As more code becomes available, not only the temporal safety properties that were already verified over the previously extracted CFGs are guaranteed to still hold if the CFGs are re-extracted (and so, refined), but new properties can be established. The problem of potential *false positives*, intrinsic to sound over-approximation, can thus be alleviated through CFG re-extraction. We have designed our framework in a way that the intra-procedural analysis is preserved, as long as the implementation is not changed. Therefore, the incremental analysis upon the arrival of previously unavailable code produces a refined model due to the fewer over-approximations w.r.t. exceptional flow.

5 The ConFlEx Tool

In this section we describe the implementation of the CFG extraction algorithms described in Section 4. First we describe some practical aspects of the implementation, and then provide experimental data that validate our tool.

5.1 Implementation

We have implemented both the algorithm for complete and for incomplete programs as the *Control Flow Extractor* tool (ConFlEx). It is based on Sawja [12], a library for the static analysis of Java bytecode. We have tailored SAWJA to address our needs. First, we have instrumented the BIR transformation to soundly provide the possible exceptions raised explicitly by `throw` instructions.

Moreover, SAWJA supports only the analysis of complete programs. Thus, we have lifted it to support open environments. On top of it, we have implemented the check of the refinement relation. Missing methods and their interfaces are provided as dummy methods with annotations. We have defined a template in Java annotation, named `GhostComponent`, to represent the interface of missing methods. Figure 1a shows the source code of an annotated missing method. It declares that the method `even` may call `odd`, or itself, and may not propagate exceptions. Here the keyword `any` denotes the set of all exception types. After compiling, the annotation is accessible as meta-data in the JBC `.class` file.

Finally, we have implemented the extraction rules from the BIR representation, as in defined in Figure 4. As described in Section 4.2, the intra-procedural analysis always produces the same set of triples if a method's implementation is not altered. Thus, we have implemented the caching of edges produced in the intra-procedural analysis. The caching allows us to perform the incremental extraction of the newly arrived component. Still, the inter-procedural analysis has to be recomputed.

5.2 Experimental Results

We validate our tool by using real-world Java applications to emulate incomplete Java bytecode systems. We choose three large, existing complete JBC applications, and replace the implementation of some of the classes with annotated methods. Then, we re-introduce the implementations incrementally, to mimic the arrival of code.

In the initial configuration, we replace the implementations of the methods of four classes with annotated methods. We perform the analysis of the resulting incomplete environment and cache the intra-procedural analysis. Next, we refine the incomplete program by re-inserting three of the four classes removed in configurations 2 and 3. For the former we reuse the cached results from configuration 1, while for the latter we perform a completely new analysis, for the purposes of assessing the impact of caching intra- results. Then, configuration 4 represents the completion of the incomplete system from set 2. The next two configurations 5 and 6 are performed over the original closed programs, with *MCA*, and *RTA* to investigate the impact of the chosen VMC resolution algorithm on the size of the resulting CFGs. Table 1 shows the experimental data. All tests have been made on an Intel i3 2.27 GHz with 4GB of RAM.

We can draw several conclusions from the experimental results. First, we observe that the number of unavailable components has a significant impact on the size of the over-approximations. For instance, configuration 1, where four classes are missing and thus has fewer instructions, produces larger CFGs than configurations 2 and 3, where a single class is missing. This can be explained partially by the excessive over-approximation of the exceptional control flow.

Table 1. Experimental results for ConFlEx

Configuration	VMC	Reused results	Missing classes	# of JBC instructions	# of Nodes	# of Edges	Time (ms) Intra	Time (ms) Inter
`Jasmin`								
1	*MCA*	no	4	25440	53467	54285	1256	339
2		yes	1	30377	35684	36228	291	109
3		no	1	30377	35684	36228	1540	104
4		yes	0	32223	34411	35052	49	104
5		no	0	32223	34411	35052	1554	85
6	*RTA*	no	0	30930	27267	27717	690	35
`Java-Cup`								
1	*MCA*	no	4	30042	76511	77345	1799	512
2		yes	1	33354	76798	77649	567	427
3		no	1	33354	76798	77649	2098	518
4		yes	0	35422	45455	46328	66	151
5		no	0	35422	45455	46328	2126	141
6	*RTA*	no	0	32049	32097	32509	983	45
`JFlex`								
1	*MCA*	no	4	52336	118414	119868	6396	877
2		yes	1	55972	77174	78678	960	631
3		no	1	55972	77174	78678	7227	407
4		yes	0	60417	72154	73175	115	181
5		no	0	60417	72154	73175	7219	177
6	*RTA*	no	0	53474	53956	54777	1676	76

Next, we see that the choice of VMC resolution algorithm has a serious impact on the CFG size. For example, in the analysis of the complete `JFlex`, *MCA* (configuration 5) produces 43% more nodes as compared to *RTA* (configuration 6). One reason is that *RTA* performs reachability analysis and eliminates dead code, and thus, the extraction is performed over fewer instructions. Further, a more precise estimation of receivers to virtual calls results in fewer call edges. Consequently, fewer nodes and edges relate to potentially propagated exceptions.

The caching of intra-procedural analysis, and consequent incremental extraction, leads to significant speed-up when compared to a whole new analysis. Also, the fixed-point computation in the inter-procedural analysis proves to be light-weight in practice, and contributes to a small fraction of the total time. This makes ConFlEx suitable for extracting CFGs in a context where the verification must be light-weight, such as in the ATM example mentioned in Section 3.

We do not provide comparative data with other extraction tools, such as Soot [16] or Wala [14] because this would demand the implementation of similar extraction rules from their intermediate representations. However, experimental results from Sawja [12] 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, ConFlEx clearly benefits from using Sawja and BIR. Also, to the best of our knowledge, ConFlEx is the first control flow analysis tool that supports incremental CFG extraction.

6 Related Work

The present work combines several aspects of program analysis, namely *soundness* w.r.t. sequences of method invocations and exceptions, *precision* w.r.t exceptional flow, and *modularity* and *incrementally* of the analysis of JBC. To the best of our knowledge, no previous work has addressed all these aspects together.

The present algorithm is modular in its essence. It analyzes components individually, as long as the interfaces for the missing components are provided. This strategy is described by Cousot and Cousot [5], and called *separate analysis*. However, a "pure" modular analysis, in the sense that each component is analyzed in isolation, would not take advantage of the inter-dependencies among the available components, and can lead to excessive over-approximation of the exceptional flow. In our case, we take inter-dependencies into account, and the isolated analyses are made incrementally.

Bandera [9] is a pioneering tool to generate abstract models from Java source programs. It is built on top of the Soot framework [16], and uses its intermediate language Jimple, in a similar fashion as ConFlEx uses Sawja and BIR. It provides several features, such as output for multiple model checkers, and some static analyses. In comparison to ConFlEx, Bandera is a versatile tool, which provides an integrated framework to program checking. However, it cannot analyze incomplete programs, and it does not address exceptional flows.

Dagenais and Hendren [6] present *partial program analysis* (PPA), a technique to build a typed intermediate representation from an incomplete program. It has been implemented in Soot, and also uses Jimple as its IR. The technique performs other analysis than control flow. Also, it is less restrictive and does not constrain the class hierarchy. However, it is admittedly unsound. Wala [14], another framework for the analysis of JBC, can also analyze partial programs. However, it ignores any side-effects from calls to unavailable methods. Thus, it is also unsound.

Ali and Lhotk [1] present a modular algorithm to generate call graphs from applications, without analyzing the API for possible call-backs. They assume that the API was coded in separation, and does not have knowledge about the application. Thus, call-backs are only possible to the application methods that overwrite a method from the API. Unfortunately this assumption is not valid for unavailable components, since developers have full knowledge of the application. The authors validate their algorithm empirically over a set of benchmarks. Thus, there is no formal argument about the soundness of their approach.

Several works propose different exception analyses. Our algorithm follows the approach of Jo and Chang [15] to extract CFGs by decoupling the intra- and inter-procedural analyses of exceptional control flow. However, they do not discuss implicit exceptions, nor address virtual method calls. Li *et al.* [17] present a framework for the extraction of CFGs *and* the model-checking of exceptional safety properties. The CFG extraction does not compute inter-procedural exceptional flow; instead, it uses a model checker to traverse the state-space. This approach requires exploration to be bounded, and is thus unsound.

7 Conclusion

We have presented a framework to extract control flow graphs from the available components of incomplete Java bytecode programs. It generalizes a previous algorithm for complete JBC programs that is defined through a transformation into an intermediate representation, and has been proven to produce sound CFGs, simulating the JVM behavior of the original programs. Our algorithm is modular in its essence. However, for higher precision, we perform the analysis of all available components together, and support the incremental refinement of the extracted CFGs as more components become available. The extracted CFGs are proven to be sound w.r.t. sequences of method invocations and exceptions. The extracted models are thus suitable for several program analyses, in particular model-checking of temporal control flow safety properties.

We have implemented the framework as the ConFlEx tool. The experimental results show that the over-approximations necessary to generate sound models (in the presence of unavailable components) have a considerable impact on the size of the extracted control flow graphs. Moreover, the over-approximations may give rise to false positive reports. ConFlEx alleviates this by providing support for the incremental refinement of the extracted models, as soon as more code becomes available. This shows the utility of ConFlEx to generate sound CFGs for incomplete programs with few missing components.

Future Work. Our framework constrains the components and how they relate w.r.t. the class hierarchy, and is limited to programs for which we know all components in advance. Our goal is to extend our analysis to truly open Java bytecode programs, where any number of components may be added in some regulated fashion. One idea is to follow the idea of *lazy parsing*, as introduced in [8]. There, instead of bounding *a priori* the unavailable components of a system, the analysis generates the constraints that the unavailable components have to fulfill to guarantee the soundness of any previous analyses.

Acknowledgments. We thank Musard Balliu, Roberto Guanciale and Siavash Soleimanifard for their valuable comments.

References

1. Ali, K., Lhoták, O.: Application-only call graph construction. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 688–712. Springer, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-31057-7_30`
2. Amighi, A., de Carvalho Gomes, P., Gurov, D., Huisman, M.: Sound control-flow graph extraction for java programs with exceptions. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 33–47. Springer, Heidelberg (2012)
3. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA, pp. 324–341 (1996)

4. de Carvalho Gomes, P., Picoco, A.: Sound extraction of control-flow graphs from open java bytecode systems. Tech. rep., KTH Royal Institute of Technology (2012), http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-104076
5. Cousot, P., Cousot, R.: Modular static program analysis. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 159–178. Springer, Heidelberg (2002)
6. Dagenais, B., Hendren, L.: Enabling static analysis for partial java programs. SIGPLAN Not. 43(10), 313–328 (2008)
7. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
8. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. The Journal of Logic and Algebraic Programming 79(7), 578–607 (2010), The 20th Nordic Workshop on Programming Theory (NWPT 2008)
9. 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 2001, pp. 177–187. IEEE Computer Society, Washington, DC (2001)
10. Freund, S.N., Mitchell, J.C.: A type system for the Java bytecode language and verifier. J. Autom. Reason. 30, 271–321 (2003)
11. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
12. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 92–106. Springer, Heidelberg (2011)
13. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
14. IBM: T.J. Watson Libraries for Analysis (2012), http://wala.sourceforge.net/
15. Jo, J.-W., Chang, B.-M.: Constructing control flow graph for java by decoupling exception flow from normal flow. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3043, pp. 106–113. Springer, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-24707-4_14
16. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop, Galveston Island, TX (October 2011)
17. Li, X., Hoover, H.J., Rudnicki, P.: Towards automatic exception safety verification. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 396–411. Springer, Heidelberg (2006)
18. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The java virtual machine specification. java se 7 edition. Tech. Rep. JSR-000924, Oracle (2012)