Sound Extraction of Control-Flow Graphs from open Java Bytecode Systems

Pedro de Carvalho Gomes, Attilio Picoco and Dilian Gurov
KTH Royal Institute of Technology, Stockholm, Sweden
{pedrodcg,picoco,dilian}@kth.se

Abstract. In the current work we present a framework to extract control-flow graphs from open Java Bytecode systems in a modular fashion. Our strategy requires the user to provide interfaces for the missing components. First, we present a formal definition of open Java bytecode systems. Next, we generalize a previous algorithm that performs the extraction of CFGs for closed programs to a modular set-up. The algorithm uses the user-provided interfaces to resolve inter-dependencies involving missing components. Eventually the missing components will arrive, and the open system will become closed, and can execute. However, the arrival of a component may affect the soundness of CFGs which have been extracted previously. Thus, we define a refinement relation, which is a set of constraints upon the arrival of components, and prove that the relation guarantees the soundness of CFGs extracted with the modular algorithm. Therefore, the control-flow safety properties verified over the original CFGs still hold in the refined model.

We implemented the modular extraction framework in the ConFlEx tool. Also, we have implemented the reusage from previous extractions, to enable the incremental extraction of a newly arrived component. Our technique performs substantial over-approximations to achieve soundness. Despite this, our test cases show that ConFlEx is efficient. Also, the extraction of the CFGs gets considerable speed-up by reusing results from previous analyses.

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 work we extract CFGs from Java bytecode (JBC). The analysis of JBC is not trivial because of the complex semantics of the language. Despite being an executable language, JBC contains several features of an object-oriented
programming language. For instance, virtual method calls (VMC) and exceptions impose challenges to the control flow analysis. Moreover, JBC is stack-based, in contrast to the usual register-based languages. Thus, the standard techniques for the analysis of executable code cannot be employed. This creates additional overhead, especially when analyzing exceptions explicitly raised by the user.

The analysis of control flow is even harder for incomplete 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 programs under-development and programs 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 executable code. The analysis of incomplete JBC programs inherits all the complications described above, in addition to the ones arising from 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 invocation.

In this paper we describe a technique for the generation of CFGs from the available components of incomplete JBC programs. We generalize a previous algorithm from Amighi et al. [2] for complete JBC programs that uses a transformation into an intermediate bytecode representation (BIR) [8]. The transformation into BIR allows the precise estimation of implicit (e.g., division by zero) and explicit (with athrow instruction) exceptions. The inter-dependencies involving components whose implementation is not yet available are captured by means of user-provided interfaces. The granularity of software components is chosen to be the one of methods. 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.

We define 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, and 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. to the JBC behavior (as defined by the JVM), as long as the specified constraints are respected. Therefore, already established behavioral or structural global properties are thus guaranteed to still hold.

The sound analysis of incomplete programs may lead to excessive over-approximation, especially of the exceptional control flow. Thus, valid global properties may fail to be established, giving rise to so-called false negatives. The extraction 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 can
be re-checked over the refined CFGs. We formalize the notion of refinement of incomplete JBC programs, and show that CFG extraction is monotone w.r.t. refinement.

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 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.

We now present our running example, which is used along the text to illustrate our definitions, and to motivate our work.

```java
public class EvenOdd {
    public static void main(String[] argv) {
        EvenOdd myobj = new EvenOdd();
        myobj.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 ***/
    public boolean even(int n);
}
```

Fig. 1: Java source program with one unavailable method

**Example 1 (Incomplete Program).** Figure 1 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 implementation of method `even` is missing, but will eventually become available. The method `odd` is available, and potentially throws an `ArithmeticException`.

The method `main` calls `parseInt` to convert the input string into an integer, then it 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 during program analysis.

Suppose we want to verify certain global properties over the available code. For example, let the property $\phi_1$ be defined informally as “if an `ArithmeticException`
is raised within a method, it must be either caught locally, or by the immediate callee method”, and $\phi_2$ be the same property, but for an `ArrayStoreException`. In the next sections we show how such properties can be verified using CFGs generated by the framework and tool described in this paper.

Outline The report is organized as follows. Section 2 motivates this work by presenting a compositional verification method that benefits from the models extracted by our technique. Sections 3, 4 and 5 summarizes results from previous works, which are necessary for the comprehension of our work. Section 6 presents the extraction algorithm for closed Java bytecode systems, and presents its correctness argument. Section 7 presents a formal framework to represent open Java bytecode systems, defines an extraction algorithm for this set-up, and proves its correctness. Section 8 describes the implementation of the extraction algorithm as the ConFlEx tool, and presents experiment results. Section 9 discusses the related work, and compares them to our approach. Finally, Section 10 summarizes our work and results, and cites possible future work.

2 Motivation: Compositional Verification

The motivation behind the present work is to support the formal verification of incomplete Java bytecode programs. Typical scenarios giving rise to such programs are ones that depend on third-party software to execute, or programs under-development. Two examples are an ATM system that is dependent on the code from users’ smart-cards, or an ERP system (such as OpenBravo [18]), which is plug-in based. It is desirable that the available components are checked against the 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 the plug-in is loaded.

For such incomplete programs, verification techniques have been developed that allow the global correctness of the program to be verified. One of these is the compositional verification technique developed by Gurov et al. [12, 11]. There, every unavailable software component is annotated with an interface declaring the provided and required methods, and a local temporal specification used to compute a so-called maximal model. The latter model simulates the behaviour 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, the component can be instantiated with this implementation, and the verified global properties will be guaranteed to hold. Such a decoupling of the verification of global properties from the actual implementations of certain components allows the verification of systems where the component implementations are not available or evolve frequently.

The correctness of the verified temporal safety properties is only guaranteed for models that are sound w.r.t. this class of properties. The behavior of the extracted models has thus to over-approximate the actual program behavior,
potentially giving rise to false negatives. To alleviate this problem, we aim at a
model extraction strategy that is *incremental*: whenever more code arrives, the
existing model can be refined, and the false negative may now be provable.

Let $\psi_u$ be the specification of a (yet unavailable) software component $u$, and
$I_u$ its interface. Let the maximal control flow graph for the pair be denoted by $\text{Max}(\psi_u, I_u)$. Also, let $G'$ be the composed control flow graph for the remaining
components, $G_u$ be the control flow graph for $u$ once it is instantiated, and $\phi$ be
a global property. The compositional verification principle can be presented as
the following proof rule:

$$
\frac {G' \uplus \text{Max}(\psi_u, I_u) \models \phi \quad G_u \models \psi_u} {G' \uplus G_u \models \phi}
$$

It states that the program’s control flow graph satisfies a global property if this
is also the case for the composition of the maximal CFG of component $u$ with
the CFGs from the remaining components; in addition, the CFG of component $u$
must satisfy its local specification, once it becomes available. Notice that the rule
can be applied consecutively w.r.t. every unavailable component, thus completely
relativizing the verification on local properties of unavailable components.

The compositional principle has been implemented as CVPP [16], a tool-
set for the compositional algorithmic verification of JBC programs. CVPP is
wrapped by PROMOVER [28], a tool which encapsulates the verification steps,
and provides a push-button interface to the user. In CVPP/PROMOVER, the
granularity of components is the one of methods. Both the global property of
the system, and the local properties of the unavailable components are provided
in temporal logic (LTL), and verified by model checking. The tool-set can verify
both structural (and thus, finite-state) and behavioral (and thus, infinite-state)
temporal properties.

*Example 2.* Consider the incomplete program from Example 1 and the men-
tioned global properties. We informally define method `even`’s interface $I_{even}$ as
“`even` may call `odd` or itself, and it cannot propagate `ArithmeticException`”,
and the method’s local property $\psi_{even}$ as “after calling `odd`, `even` must terminate
normally”.

First we want to check whether the global property $\phi_1$ holds. We construct
the maximal CFG for $\psi_{even}$ and $I_{even}$, compose it with the CFGs of `main` and
`odd`, and check $\phi_1$. The property turns out to hold, and once the implementation
of `even` is provided, we simply extract its CFG, and check it against the local
property $\psi_{even}$. Again, the property holds, and hence the correctness of the
program is established w.r.t. $\phi_1$.

Next, we want to verify $\phi_2$ over the same composed model, created with the
maximal CFG from $\psi_{even}$ and $I_{even}$. However, $\phi_2$ does not hold since neither
the interface, nor the local property restrict an `ArrayStoreException` from be-
ing raised by `even`. Still, it turns out to be a false negative: after the code of `even`
becomes available and all previous CFGs are refined, the property can be
established to hold.
3 Formal Java virtual machine framework

In this section we present an overview of the formal Java virtual machine framework defined by Freund and Mitchell [10]. The work considers a significant fragment of the Java bytecode instructions set, which captures most of challenges on its static analysis. Specifically, virtual and interface method calls, and exceptions are featured. We summarize such definitions, focusing only in the relevant aspects to the control-flow analysis.

A compiler that targets Java bytecode generates class files, one for each declared class, or interface. Each class declaration contains a symbolic name, type information, and the declaration of its method and fields. Let Class-Name and Interface-Name be the (countably) infinite sets of all class and interface names, respectively. Bytecode programs use method references, interface method references and field references to identify methods, interface methods and fields. These references are triples which describe the method (or interface method) in which it was declared, the method (or field) signature, and its type. They are generated from the grammar in Figure 2.

![Fig. 2: Grammar generating references](image)

- **Method-Ref**: $\{\text{Class-Name}, \text{Label}, \text{Method-Type}\}_M$
- **Interface-Method-Ref**: $\{\text{Interface-Name}, \text{Label}, \text{Method-Type}\}_I$
- **Field-Ref**: $\{\text{Class-Name}, \text{Label}, \text{Field-Type}\}_F$

In this work we consider a subset of the JVML$_f$ instruction set described in [10]. Although it is significantly smaller, the subset contains one representative for each of the distinct cases to analyze statically the control-flow. For example, we have omitted the invokeinterface instruction, since the control-flow analysis for its case is analogous to invokevirtual. The instructions jsr q and ret r for subroutine are not considered because they are deprecated since the JBC version 1.6 [24]. Figure 3 shows the bytecode instructions set considered in our project. We use the symbol $x$ to denote a local method variable, and $p$ to an instruction address.

Java bytecode is a stack-based executable language. That is, most of the operands for its instructions are stored in the operand stack. For example, the if $x$ instruction branches to position $x$ if the value on the top of the stack is zero. Also, the exception being raised by the athrow instruction, or the object whose method is being called by the invokevirtual, are also on top of the operand stack.

The JBC semantics models a program as an environment, as in most semantics frameworks. Figure 4 shows the definition of an environment $\Gamma$, which is the union of the partial mappings from classes and interfaces names, and method references, to their respective definitions. A class is defined by its parent class, the set of interfaces it implements, and its fields. One interface contains the set
\textit{Instruction ::= \texttt{nop} | \texttt{push c} | \texttt{pop} | \texttt{dup} | \texttt{add} | \texttt{div} |
| \texttt{if p} | \texttt{goto p} |
| \texttt{load x} | \texttt{store x} |
| \texttt{new Class-Name} |
| \texttt{athrow} |
| \texttt{getfield Field-Type} | \texttt{putfield Field-Type} |
| \texttt{invokespecial Method-Ref} |
| \texttt{invokevirtual Method-Ref} |
| \texttt{vreturn} | \texttt{return} \\

Fig. 3: Subset of the JBC instructions

of interfaces it inherits from, and the set of methods it provides. A method is
defined by its array of instructions, and a list of exception handlers. An exception
handler is 4-tuple \( \langle b, e, t, \sigma \rangle \), where \([b, e)\) is the address range covered by
the handler, \( t \) is the address of the control-point which handles the exception, and
\( \sigma \in \text{Class-Name} \) is the exception type.

\begin{align*}
\Gamma^I : \quad \text{Interface-Name} & \rightarrow \left\{ \begin{array}{l}
\text{interfaces : set of Interface-Name} \\
\text{method : set of Interface-Method-Ref}
\end{array} \right. \\
\Gamma^C : \quad \text{Class-Name} & \rightarrow \left\{ \begin{array}{l}
\text{super : Class-Name} \\
\text{interfaces : set of Interface-Name} \\
\text{fields : set of Field-Ref}
\end{array} \right. \\
\Gamma^M : \quad \text{Method-Ref} & \rightarrow \left\{ \begin{array}{l}
\text{code : Instruction}^+ \\
\text{handlers : Handler}^*
\end{array} \right.
\end{align*}

\[ \Gamma = \Gamma^I \cup \Gamma^C \cup \Gamma^M \]

Fig. 4: Environment \( \Gamma \) of a Java program

The standard Java virtual machine contains a bytecode verifier (JBV), which
performs several sanity checks on the code before the execution starts. It checks
the correctness of the code format, if a method always terminates with a \texttt{return}
or \texttt{athrow} instruction, and if branches refer to valid positions, among other
analyses. The definition below states that a well-formed program is the one that
passes successfully through all the verification tasks. In this work we assume
that the input bytecode is always well-formed.

\textbf{Definition 1 (Well-Formed Java Program).} A well-formed Java bytecode
program is a closed program which passes the JVM bytecode verification. The
exhaustive list of verification task is presented in [29].
Along an execution of the JVM, an active method is represented by an activation record. This is a 5-tuple which contains the method’s reference $m$, the address $p$ of the next instruction to be executed, a map $f$ from the local variables to values, the method’s operand stack $s$, and $z$ is the information about the initialization of the object. The records are placed in the call stack, which stores in which sequence the methods are invoked. The top of the call stack contains the activation record of the current method being executed, or the record $\langle e \rangle_{exc}$, representing the case when an exception is raised. Figure 5 shows the syntax for the call stack.

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

Fig. 5: Syntax of the JVM call stack

It is important at this point to make a clear distinction between the operand stacks, and the call stack. An operand stack is defined for each method, and stores the values used by its instructions. A call stack is unique for a given JVM sequential program, and stores the records for the current active methods. In summary: a JVM execution contains a single call stack, which by its turn may contain several operand stacks.

An execution state of the Java virtual machine is defined as a configuration $C = A; h$, where $A$ is the call stack, and $h$ represents a memory heap. The JVM behavior is the infinite-state transition system where the states are all the possible configurations, and the transition relation is defined by the operational semantics of the JBC instruction set, as presented in [10].

The Java bytecode is an executable language. Nevertheless, it contains some aspects of an object-oriented programming language. One is inheritance, which is the code reusage mechanism that allows one class to extend the definitions of another existing class. An environment has the inheritance definitions in $\Gamma^C$.interfaces and $\Gamma^I$.interfaces, which contains the interfaces a class or an interface will extend, and in $\Gamma^C$.super, which tells from what parent class a child class extends. The inheritance defines a type hierarchy between classes and interfaces. Every JBC program has a class hierarchy, being the class java.lang.Object the root.

The inheritance is transitive in JBC programs. That is, one class or interface inherits in cascade from its immediate classes and interfaces. The subtyping relation, defined for two class or interfaces $\tau_1$ and $\tau_2$ holds whenever $\tau_1$ inherits transitively from $\tau_2$. We use the notation $\Gamma \vdash \tau_1 \ll{\subset} \tau_2$ to denote the a subtyping holds for a given environment. Figure 6 shows the rules for the subtyping relation.

The subtyping plays a key role in the control-flow analysis. First, because of the polymorphism, another OOP feature of bytecode. Polymorphism is possibility to have more than one implementation for the same method signature.
In JBC, it is presented as subtype polymorphism. That is, it is possible for several classes in a sub-hierarchy to have the same method signature, but with a different implementation. We call those methods as virtual.

The invocation of virtual method is executed by the `invokevirtual` instruction, which operates over two parameters. One is the `Method-Reference`, which is hard-coded in the bytecode. The `Method-Reference` declaration contains the method signature, and the `Class-Name`, which we say to be the static type of the method. However, the second parameter is on the top of the operand stack. It contains an object reference, and the dynamic type of this object is what determines which of the polymorphic method implementations will be invoked. The exact dynamic type can only be determined in run-time. The only guarantee, provided by the JBV, is that the possible dynamic types are always sub-types of the static type. Virtual method call (VMC) resolution algorithms determine statically the set of possible receivers for a given virtual invocation.

Exceptions are objects used to signal some abnormal condition during the program execution. In JBC, exceptions are objects whose class is a subtype of the `java.lang.Throwable` class. The exception classes are the ones present in the standard Java API, or user-defined. Also, an exception can either be raised explicitly by the user, or implicitly, by the erroneous execution of some instruction (e.g., division by zero). Explicit exceptions are raised with the `athrow` instruction. Its only operand is the reference to the exception to be thrown, which is on the top of the operand stack. Thus, static analysis techniques have to perform some stack evaluation to determine the possible types of exceptions.

After the raise of an exception, the JVM verifies if there exists a suitable code block to handle it. This check searches for the first handler on the method’s handler table whose address range contains the address of the control-point where the exception was raised, and its type is a sub-type of the exception. If a suitable handler is found, the control is transferred to the first instruction in that block; otherwise the current method is terminated abruptly, and the exception is propagated to the calling method, which now should handle the exception. This
process continues until one of the methods in the stack of method invocations handles the exception, or the program terminates.

4 The BIR Language

We now describe the Bytecode Intermediate Language, a stackless representation of the Java bytecode. The use of BIR language provides several advantages. First, the JVM is a stack-based machine. Thus, it requires some sort of stack analysis to determine the types of the operands. This type of analysis is not trivial, as it requires knowledge of the contents of the whole stack, while performing some operations on it. The transformation from JBC instructions into BIR generates a set of instructions that are no more stack-based. They are variable-based instead, and represent expression trees, differently from those of Java Bytecode. Next, the transformation provided generates code that usually has a smaller size than the original one. Finally, BIR also supports a subset of the Java unchecked exceptions [19]. It provides a set of instructions that perform assertions related to these exceptions.

In [8] the authors define the semantics of the BIR language. They prove that the transformation algorithm and the language semantics are correct, since they preserve the original semantics of the program, regarding the use of the relations over values, environments and observable events. Our extraction process is purely syntactic, so the correctness of the BIR semantics is unrelated to our work. However, it brings reliability to our correctness proof since the syntactic transformation from JBC into BIR is part of the proof itself.

Figure 7 shows the definitions for the syntax of the BIR language. BIR contains both local and temporary variables: the former are identifiers already defined in the Bytecode; the latter are new identifiers. It also provides expressions and instructions to handle variable and field assignments.

We must take into account the order of the object creation and of the exception throwing to define the transformation correctly. The two cases address the same problem: both orders have to be explicitly defined so they can hold, as done in the Bytecode. The former task is performed by the Java Virtual Machine in two separate steps: first, raw object allocation; then, constructor call. Only when the object is created correctly, it can be referenced and used. In a sequence of object creations, the sequence order has to be maintained. Moreover, the steps related to different objects must not overlap. This is to preserve any dependence among the objects themselves. The latter case mentioned above implies that the transformation has to check dynamically for run-time errors due to different exceptions.

Let C be a class in a Java program. BIR implements the two instructions [mayinit C] and [var, := new C(e1,...,en)] to handle the class initialization (the former), the allocation of the object and the call to its constructor (the latter). The class initialization is always performed before the others. This step occurs only once, that is, on the moment when a class is referenced for the first time, either for the creation of the object or for a static method call. The excep-
Fig. 7: Expressions and Instructions of BIR

Fig. 8: BIR assertions, and the associated unchecked exceptions

The algorithm transforms the input JBC code into a set of BIR instructions. The function BC2BIR_{instr} is applied to each JBC instruction to perform the transformation. The transformation is defined as follows:

**Definition 2 (BIR Transformation Function).** Let AbsStack ∈ Expr*. The rules defining the instruction-wise transformation BC2BIR_{instr} : N × instr_{JBC} × AbsStack → ((instr_{BIR})^* × AbsStack) ∪ {Fail} from Java Bytecode into BIR are given in Figure 9.

A key point of the algorithm is the way to manage the operand stack, and thus stack-based code. This is done by using a symbolic stack that allows a transformation from the original code to a set of 3-address instructions. Figure 9
shows the core of the algorithm, that is, the function mapping a Bytecode instruction into a list of BIR instructions. At the same time, these instructions are symbolically executed by using this abstract stack, which refers to symbolic expressions.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>nop</td>
</tr>
<tr>
<td>push c</td>
<td>if p</td>
</tr>
<tr>
<td>dup</td>
<td>goto p</td>
</tr>
<tr>
<td>load x</td>
<td>return</td>
</tr>
<tr>
<td>add</td>
<td>vreturn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>store x</td>
<td>[x:=e] or [t_{pc}^0:=x;x:=e]</td>
</tr>
<tr>
<td>putfield f</td>
<td>[nonnull e;FSave(pc,f,as);e.f:=e']</td>
</tr>
<tr>
<td>invokevirtual m</td>
<td>[nonnull e;HSave(pc,as);t_{pc}^0:=e.m(e'<em>{1}...e'</em>{n})] or [HSave(pc,as);t_{pc}^0:=new C(e'<em>{1}...e'</em>{n})]</td>
</tr>
</tbody>
</table>

Fig. 9: BC2BIR_{inst} - Transformation of a BC instruction at pc

Most of the instructions modify the abstract stack when they are symbolically executed. The transformation of return and jump instructions is simple, as well as that of [nop] instructions. The transformation of a subset of instructions, like [load x] and [push c], do not produce any BIR instruction, instead. The use of temporary variables (t_{pc}^i) allows to handle those instructions affecting memory locations, such as [store x], [putfield f] and [invokevirtual C.m]. These variables store each element on the stack, whose value might change. Thus, temporary variables are necessary to preserve the consistency of the operand types. The instruction [new C] for the object creation is preceded by [mayinit C] for the class initialization. The reference to the new object is then pushed onto the stack.

The expressions representing the stack elements must not depend on the control flow. The control flow path is not linear when there are branches and join points. However, the size of the abstract stack has to remain the same, whereas the actual size of its content may vary during the transformation. The proposed solution is the definition of a normalized stack containing temporary variables that store the original stack elements.

Example 3 (JBC and BIR Representation). Figure 11 shows the JBC and BIR versions of method odd() from Figure 3. 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 [nonnull] instruction asserts that NullPointerException can potentially be raised at this program point.
5 Program Models

Control-flow graphs are an abstract model of a program. To define the structure and behavior of a CFG we follow Gurov et al. and use the general notion of model [12][15].

Definition 3 (Model, Initialized Model). A model is a (Kripke) structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ where $S$ is a set of states, $L$ is a set of labels, $\rightarrow \subseteq S \times L \times S$ a labeled transition relation, $A$ a set of atomic propositions and $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation assigning the set of atomic propositions that hold on each state $s \in S$. An initialized model is a pair $(\mathcal{M}, E)$ with $\mathcal{M}$ a model and $E \subseteq S$ a set of entry states.

Method graphs are the basic building blocks of control-flow graphs. Let Method-Ref be the infinite set of all possible method signatures, and Excp-Name $\subseteq$ Class-Name be the infinite set of all exceptions classes in Java. We define a method graph for sequential programs with procedures and exceptions as the instantiation of an initialized model, as follows.

Definition 4 (Method Graph). A method graph for method $m \in$ Method-Ref over sets $M \subseteq$ Method-Ref and $E \subseteq$ Excp-Name is an initialized model $\mathcal{G}_m = (\mathcal{M}_m, \mathcal{E}_m)$, where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a model 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

\hspace{1em} Fig. 10: Comparison between JBC and BIR representation

<table>
<thead>
<tr>
<th>Java bytecode</th>
<th>BIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: iload x</td>
<td>0: if ( x &gt;= 0) goto 5</td>
</tr>
<tr>
<td>1: ifge 12</td>
<td>1: mayinit ArithmeticException</td>
</tr>
<tr>
<td>4: new</td>
<td>1: t0 := new ArithmeticException()</td>
</tr>
<tr>
<td></td>
<td>2:nonnull t0</td>
</tr>
<tr>
<td>7: dup</td>
<td>8: invokespecial ArithmeticException()</td>
</tr>
<tr>
<td>8: invokespecial ArithmeticException()</td>
<td>2: t0 := new ArithmeticException()</td>
</tr>
<tr>
<td></td>
<td>3: notnull t0</td>
</tr>
<tr>
<td>11: athrow</td>
<td>12: iload x</td>
</tr>
<tr>
<td>13: ifne 18</td>
<td>16: ireturn</td>
</tr>
<tr>
<td>14: load0</td>
<td>17: ireturn</td>
</tr>
<tr>
<td>15: return 0</td>
<td></td>
</tr>
<tr>
<td>18: aload 0</td>
<td></td>
</tr>
<tr>
<td>19: iload x</td>
<td></td>
</tr>
<tr>
<td>20: iconst 1</td>
<td></td>
</tr>
<tr>
<td>21: isub</td>
<td></td>
</tr>
<tr>
<td>22: invokevirtual even(int)</td>
<td>8: t0 := this.even(x - 1)</td>
</tr>
<tr>
<td>23: ireturn</td>
<td>9: return t0</td>
</tr>
</tbody>
</table>
node is tagged with the method signature it belongs to and with at most one exception). \( \mathcal{E}_m \subseteq V_M \) is the (non-empty) set of entry control points of \( m \).

A method graph represents the control-flow structure of a method. On it, nodes represent the control points of the method, and transitions represent the transfer of control between the control points. The set \( \mathcal{E}_m \) contains the node relative to the entry point of a method. Nodes tagged with the atomic proposition \( r \) represent return control-points. A node can be either normal, having no exception as atomic proposition, or exceptional, having exactly one exception. The transitions are labeled either by a method signature (denoting a method call), or by \( \varepsilon \) (to denote invisible actions).

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

**Definition 5 (Control-Flow Graph Interface).** A Control-Flow Graph interface is a triple \( I = (I^+, I^-, I^e) \), where \( I^+, I^- \subseteq \text{Method-Ref} \) are finite sets of provided and (externally) required method signatures, respectively. \( I^e \subseteq I^+ \times E \) is the set 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. The Interface composition is defined as \( I_1 \cup I_2 = (I_1^+ \cup I_2^+, (I_1^- \cup I_2^-) \setminus (I_1^+ \cup I_2^+), I_1^e \cup I_2^e) \).

Let \( \sqcup \) denote the standard disjoint union of two initialized models. We define a method’s control-flow graph as pair of its method graph and interface, and the composition of two control-flow graphs as follows.

**Definition 6 (Control-Flow Graph Structure).** A Control-Flow Graph \( G \) with interface \( I \), written \( G : I \) is inductively defined by:

- \((M_m, \mathcal{E}_m) : (\{m\}, I^-, I^e_m) \) if \((M_m, \mathcal{E}_m) \) is a method graph for \( m \) over \( I^- \) and \( I^e_m \),
- \( G_1 \sqcup G_2 : I_1 \cup I_2 \) if \( G_1 : I_1 \) and \( G_2 : I_2 \).

**Example 4 (CFG from incomplete program).** Figure 11 shows the CFG for the available methods of the incomplete Java program from Figure 1, relative to their BIR representation. The graph thus consists of the method graphs of methods \( \text{main} \) and \( \text{odd} \), with the nodes being tagged with the method’s signature and address in the code array. 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. The three are also exceptional return nodes (i.e., exceptional nodes tagged with the atomic proposition \( r \)), and indicate the propagation of the respective exception by the method.
Fig. 11: Method graphs for available methods

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 $0_{\text{NFE}, r}\text{main}$.

The interface of the CFG composed of the two method graphs is the triple $\langle \{\text{main, odd}\}, \{\text{even}\}, \{(\text{odd, ArithmeticException}), (\text{main, ArithmeticException}), (\text{main, NumberFormatException})\}\rangle$. The interface of the missing method even is $\langle \{\text{even}\}, \{\text{odd}\}, \emptyset\rangle$. It declares that the method may call itself or odd, and does not propagate any exceptions.

The structure of a closed CFG induces a behavior, which is the push-down automata used to model the JVM call stack. The Definition below extends the CFG behavior introduced in [15], to model the exceptional control-flow.

**Definition 7 (CFG Behavior).** Let $G = (M, E) : I$ be a closed flow graph with exceptions such that $M = (V, L, \rightarrow, A, \lambda)$. The behavior of $G$ is described by the initialized model $b(G) = (M_b, E_b)$, where $M_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$ s.t.:

- $S_b \in V \times V^*$, i.e., states are pairs of control node and stack of control nodes,
- $L_b = \{\tau\} \cup L^C_b \cup L^X_b$ where $L^C_b = \{m_1 l m_2 \mid l \in \{\text{call, ret, xret}\}, m_1, m_2 \in I^+\}$ (the set of call and return labels) and $L^X_b = \{l x \mid l \in \{\text{throw, catch}\}, x \in \text{EXCP}\}$ (the set of exceptional transition labels),
- $A_b = A$
The set of entry states is defined by $E_b = E \times \{\epsilon\}$, where $\epsilon$ denotes the empty sequence.

Intuitively, $\tau$-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 the 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 the control is transferred to the successor normal control node. A 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 the control is transferred to the successor exceptional node tagged with $x$.

Now we show how the induced CFG behavior models the JVM behavior. We define the abstraction function $\theta$, which maps a JVM configuration to a CFG behavioral configuration, as follows.

**Definition 8 (Abstraction Function for VM States).** Let $\text{Conf}$ be the set of JVM execution configurations and $S_b$ the set of states in $G_{bhe}$. Then $\theta : \text{Conf} \to S_b$ is defined inductively as follows:

$$
\theta(c) = \begin{cases} 
\langle \phi^p_m, \epsilon \rangle & \text{if } c = (\langle m, p, f, s, z \rangle.c; h) \\
\langle \phi^p_{m,1}, \theta(A; h) \rangle & \text{if } c = (\langle m, p, f, s, z \rangle.A; h) \\
\langle \phi^{x,r}_{m}, \theta(A; h) \rangle & \text{if } c = (\langle x \rangle_{\text{exc}}, \langle m, p, f, s, z \rangle.A; h) \\
\langle \phi^{x,r}_{m} \rangle & \text{if } c = (\langle x \rangle_{\text{exc}}.c; h)
\end{cases}
$$
Function $\theta$ is defined recursively, and applies to all activation records on the call stack. The symbol $♭$ denotes the special abort control-point, which is reached only when the call stack is empty, caused by an uncaught exception.

**Example 5 (CFG Behavior).** Let’s suppose that the implementation of method `even` is provided, with the respective method graph being the one from Figure 12. The composition of this, with the method graphs from Figure 11 results in a closed CFG structure.

![Fig. 12: CFG for even method](image)

Following is an example run through the (infinite-state) behavior induced by the closed CFG:

\[
(v_1, \epsilon) \xrightarrow{\tau} \text{even} (v_2, \epsilon) \xrightarrow{\tau} \text{even} (v_3, \epsilon) \xrightarrow{\tau} \text{even} (v_4, \epsilon) \xrightarrow{\text{main call even}} \text{even} (v_{15}, v_4) \xrightarrow{\tau} \text{even}
\]

\[
(v_{16}, v_4) \xrightarrow{\tau} \text{even} (v_{17}, v_{18} \cdot v_4) \xrightarrow{\text{even call odd}} \text{odd} (v_8, v_{18} \cdot v_4) \xrightarrow{\tau} \text{odd} (v_{19}, v_{18} \cdot v_4) \xrightarrow{\text{throw A.E.}} \text{odd}
\]

\[
(e_3, v_{18} \cdot v_4) \xrightarrow{\text{odd xreturn even}} \text{even} (e_4, v_4) \xrightarrow{\text{catch A.E.}} \text{catch A.E.} (v_{20}, v_4) \xrightarrow{\tau} \text{even} \ldots
\]

This sample represents an execution starting in the entry control node of the `main` method, next invoking `even`, and then `odd`. An `ArithmeticException` 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.
6 Extraction algorithm for closed programs

Our modular CFG algorithm is based on a previous result from Amighi et al. [2,3], which is designed for closed programs only. They proposed an indirect algorithm that first translates the JBC program into the BIR language, and next extracts the CFGs. Their algorithm extract CFGs that induce CFG behaviors that are proven to simulate the JVM behavior. We also reuse this correctness results to establish the soundness of our modular algorithm.

We now describe the extraction algorithm for closed BIR programs. Let \( \Gamma \) be the BIR environment for a closed program, \( \Gamma[m].\text{code} \) be the instructions array for some method \( m \), and \((pc, i)\) be some BIR instruction \( i \) in the position \( pc \) in the array. The control flow graph extraction function is defined as follows.

**Definition 9 (Control Flow Graph Extraction).** The instruction-wise extraction function \( \mathcal{G} : (\text{Method-Ref} \times \text{Instr} \times \mathbb{N}) \rightarrow \mathcal{P}(V \times L_m \times V) \) is defined by the rules in Figure 13. The method graph for \( m \) is defined as \( \mathcal{G}_m = \bigcup_{(pc, i) \in \Gamma[m].\text{code}} \mathcal{G}^{pc,i} \). The control flow graph for the complete program is defined as \( \mathcal{G}(\Gamma) = \bigcup_{m \in \text{dom}(\Gamma)} \mathcal{G}_m \).

The indirect algorithm is defined by the functional composition of the BIR transformation with the extraction algorithm from Definition 9, as \( \text{BC2BIR} \circ \mathcal{G} \). Each JBC instruction in the body of \( m \) is mapped into a set of BIR instructions. Next, the whole set of BIR instructions of \( m \) is processed to produce its method graph. Finally, the control flow of the program is represented by a control-flow graph that is the union of all the method graphs for the methods in the program.

The simplest instructions are assignments, \([\text{nop}]\) and \([\text{mayinit}]\). These produce a single edge from the current control node to the normal next one. Return instructions also add a single edge to a return node, that is a node referring to the same control point, but marked with the atomic proposition \( r \). Jumps can be either conditional (instruction \([\text{if } \text{expr } \text{pc'}]\)) or unconditional (\([\text{goto } \text{pc'}]\)): the former introduce two edges (to the next control point and to \( \text{pc'} \), respectively) to represent the branch; the latter add a single edge to the node referring to the control point \( \text{pc'} \).

The \([\text{throw } X]\) and method call instructions are treated similarly as they both depend on the static type of the object the instruction is invoked on (the exception thrown and the calling object, respectively). BIR provides the static type of the object only. For \([\text{throw } X]\), let \( X \) be the set containing the static type and all of its subtypes. For any \( x \in X \), an exceptional edge is added together with an appropriate handler edge, if any, according to the exception table. For normal method calls, let \( \text{res}^\alpha \) be the set of the method receivers determined after resolving the call. It will contain the method referring to the static class type of the original object and those referring to its children classes. In this case a normal edge will be added for each element \( n \in \text{res}^\alpha \).

Assertion instructions produce a branch: a normal edge if the exception is not raised, and an exceptional edge, together with a handler edge from the exception table, if any, when the assertion fails. The \([\text{new } C]\) instruction adds only one
H_{pc,x,l}^m = \begin{cases} \{ (o_{pc,x,l}^m, l, O_{pc,x,l}^m) \} & \text{if } h_{pc,x}^m = \text{undef} \\ \{ (o_{pc,x,l}^m, l, O_{pc,x,l}^m), (o_{pc,x,l}^m, \varepsilon, O_{pc,x,l}^m) \} & \text{if } h_{pc,x}^m = pc' \end{cases}

\Lambda_{pc,n}^m = \bigcup_{(x|^{n\in\text{Exception}})} H_{pc,x,n}^m

\mathcal{G}_{pc,i}^m =
\begin{cases}
\{ (o_{pc,i}^m, \varepsilon, o_{pc,i+1}^m) \} & \text{if } i \in \text{Assignment} \cup \{ [\text{nop}], [\text{mayinit}] \} \\
\{ (o_{pc,i}^m, \varepsilon, o_{pc,i+1}^m), (o_{pc,i}^m, \varepsilon, o_{pc,i'}^m) \} & \text{if } i = [\text{if expr pc'}] \\
\{ (o_{pc,i}^m, \varepsilon, o_{pc,i+1}^m) \} & \text{if } i = [\text{goto pc'}] \\
\{ (o_{pc,i}^m, \varepsilon, o_{pc,i+1}^m) \} & \text{if } i \in \text{Return} \\
\bigcup_{(x|^{x<:X})} H_{pc,x,i}^m & \text{if } i = [\text{throw X}] \\
\{ (o_{pc,i}^m, \varepsilon, o_{pc,i+1}^m) \} \cup H_{pc,i}^m & \text{if } i \in \text{Assertion} \\
\{ (o_{pc,i}^m, C, o_{pc,i+1}^m) \} \cup H_{pc,i}^{\text{Exception}} \cup \Lambda_{pc,n}^m & \text{if } i \in \text{NewObject} \\
\bigcup_{n \in \alpha(n)} \{ (o_{pc,i}^m, n, o_{pc,i+1}^m) \} \cup \Lambda_{pc,n}^m & \text{if } i \in \text{MethodCall} 
\end{cases}

Fig. 13: CFG extraction rules from BIR

normal edge together with an exceptional edge because of a *NullPointerException*.

The propagation of a given exception is performed backward according to the potential sequence of invocations. The function \( \Lambda_{pc,n}^m \) adds exceptional edges about the considered exception to those program points where the caller method invokes the method propagating the exception. The function \( H_{pc,x}^m \) checks if there is a suitable handler for the given exception. If there is, an edge is added to it. Otherwise the edge is added to an exceptional return node, and the propagation continues, until either a handler is found for it or there are no more methods in the sequence considered.

The implementation follows the strategy presented in [20]: the whole control flow containing both normal and exceptional data is extracted in two separate phases. The analysis is actually performed in two steps: first, an intra-procedural analysis; then an inter-procedural one. The former builds the graph for each method, according to the extraction rules. The latter manages the propagation of exceptions among the method calls.

7 Modular CFG Extraction

We have presented in Section 3 a formal JVM framework, and in Section 6 we described a sound CFG extraction algorithm, which is proven to preserve the JVM behavior. Those definitions are valid for closed systems only. That is, they are applicable only for software systems whose all components are provided.
However, there are scenarios where at least one of the components is not available, but still one wants to analyze the available components. Some common situations are systems under development, or a software system which depends on a third-part component. In both scenarios one may still extract CFGs and verify properties from the available components.

In this section we generalize the previous definitions to open Java bytecode systems. First, we extend the formal JVM framework to represent unavailable software components. The missing components are represented by user-provided interfaces. Next, we generalize the previous algorithm to modularly extract CFGs from the available components. That is, the algorithm extracts the method graphs for the available methods, and resolve the inter-dependencies involving missing methods by using the provided interfaces.

Eventually the missing components will arrive, and the open system will become close, and can execute. However, the arrival of a component may affect the soundness of CFGs which have been extracted previously. Thus, we define the refinement relation, which are constraints over the arrival of components, to guarantee that soundness is preserved. We conclude by proving that if the refinement relation holds, then the CFGs extracted with the modular algorithm from an open system are sound over-approximations for any closed system assembled from it. Therefore, the safety properties verified over the CFGs still hold.

7.1 The open extraction algorithm

We now generalize the previous definitions for open Java bytecode programs. We call a program open if the implementation of at least one of its components is unavailable. Note that open JBC programs always result in open BIR programs.

\[
\Gamma_o^M : \text{Method-Ref} \rightarrow \left\{ \begin{array}{l}
\text{code : Instruction}^*, \\
\text{handlers : Handler}^*
\end{array} \right\}
\]

\[
\Gamma_o = \Gamma^I \cup \Gamma^C \cup \Gamma_o^M
\]

Fig. 14: Open Java bytecode environment

Figure 14 presents the modified definition of open Java bytecode environment, importing the unchanged definitions for Java interfaces and classes (\(\Gamma^I\) and \(\Gamma^C\), respectively) from Figure 4 for closed programs. The main difference is that the instruction array of a method may now be empty; we use \(\text{impl}_{\Gamma_o}(m)\) to denote \(\Gamma_o^M.\text{code} \neq \epsilon\), i.e., that method \(m\) has an implementation in environment \(\Gamma_o\). Further, the definition of \(\Gamma_o^M[m].\text{handlers}\) has a different interpretation from \(\Gamma^M[m].\text{handlers}\); here, it represents constraints on the implementation of the unavailable method. Every tuple in \text{handlers} determines one exception type that cannot be propagated by the method, and are provided in the user-defined interface.
One important aspect of the definition of open environment is that it contains all information about the type hierarchy. As discussed in Section 3, an exception in a Java program is an object whose type is a subtype of `java.lang.Throwable`. Thus, we can estimate the set of exceptions types from a given open environment, as follows.

**Definition 10 (Exceptions Set).** Let $\Gamma_o$ be an open Java bytecode environment. We define $E_{\Gamma_o} = \{c \in \text{Excp} \mid \Gamma_o \vdash c : \text{java.lang.Throwable}\}$ as the set of exception types in this environment.

We now extend the Definition 9 to a modular set-up. There are two modifications we need to make.

The first one is w.r.t. to virtual method call resolution. In contrast to the definition for closed programs, it cannot be parametrized on an arbitrary VMC algorithm, since any VMC algorithm that relies on code analysis may provide unsound estimation in the absence of code of receivers to virtual method calls, since it cannot determine which classes are referenced. We therefore fix the VMC resolution algorithm to our Modular Class Analysis (MCA), a relaxation of CHA to soundly over-approximate the possible receivers to the methods with the same signature, from sub-types of the static type. MCA considers both available and missing methods. However, only the methods that are on the domain of $\Gamma^M$ are considered. Thus, a missing method that will overload an inherited method should be defined in the open environment.

The second modification concerns the $N$ function 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 the exception types, excluding those declared in the user-provided interface to be never propagated.

Let $\Gamma_o$ be an open BIR environment, $\text{dom}$ be the function that returns the domain of a partial function, and $(\text{pc}, i)$ be some BIR instruction $i$ in the position $\text{pc}$ in the array. The control flow graph extraction function is defined as follows.

**Definition 11 (Modular Extraction Function).** The instruction-wise extraction function $oG : (\text{Method-Ref} \times \text{Instr} \times \mathbb{N}) \rightarrow \mathcal{P}(V \times L_{m} \times V)$ is defined by the rules in Figure 15. The method graph for $m$, such that $\Gamma_o[m].\text{code} \neq \epsilon$, is defined as $oG_m = \bigcup_{(\text{pc},i) \in \Gamma_o[m].\text{code}} oG^{|i|}_{\text{pc}}$. The control flow graph for the open program is defined as $oG(\Gamma_o) = \bigcup_{\{m \mid \Gamma_o[m].\text{code} \neq \epsilon\}} oG_m$.

7.2 Correctness Proof

The main purpose of the transformation above is to enable the extraction of CFGs from the available components of incomplete JBC programs. The extracted CFGs can be used to verify global temporal safety properties as explained in Section 2. Further, the transformation allows the extracted CFGs
MCA(\text{c.ns}) = \begin{cases} \{ \text{c.ns} \mid \text{c} <: \text{C} \land \text{c.ns} \in \text{dom}(\Gamma^M) \} & \text{if call is virtual} \\ \{ \text{c.ns} \} & \text{otherwise} \end{cases}

\begin{align*}
H_{m}^{\text{pc},x,l} & = \begin{cases} \{ (\sigma_{m}^{\text{pc},x,l}, \cdot, \sigma_{m}^{\text{pc},x,l}) \} & \text{if } h_{m}^{\text{pc},x} = \text{undef} \\
\{ (\sigma_{m}^{\text{pc},x,l}, \cdot, \sigma_{m}^{\text{pc},x,l}) \} & \text{if } h_{m}^{\text{pc},x} = \text{pc}' \\
\bigcup_{x \in E} \Gamma_{m}^{\text{pc},x,l} \text{ handlers } H_{m}^{\text{pc},x,l} & \text{otherwise} \end{cases} \\
N_{m}^{\text{pc},n} & = \begin{cases} \{ (\sigma_{m}^{\text{pc},n}, \cdot, \sigma_{m}^{\text{pc},n}) \} & \text{if } \Gamma_{m}[n].\text{code} \neq \varepsilon \\
\bigcup_{x \in E} \Gamma_{m}^{\text{pc},x,n} & \text{otherwise} \end{cases} \\
G_{m}^{\text{pc},i} & = \begin{cases} \{ (\sigma_{m}^{\text{pc},i}, \cdot, \sigma_{m}^{\text{pc},i}) \} & \text{if } i \in \text{Assignment} \cup \{ \text{nop}, \text{mayinit} \} \\
\{ (\sigma_{m}^{\text{pc},i}, \cdot, \sigma_{m}^{\text{pc},i}) \} & \text{if } i = \text{[if expr pc']} \\
\{ (\sigma_{m}^{\text{pc},i}, \cdot, \sigma_{m}^{\text{pc},i}) \} & \text{if } i = \text{[goto pc']} \\
\bigcup_{x \in E} \Gamma_{m}^{\text{pc},x,i} & \text{if } i = \text{Return} \\
\bigcup_{x \in E} \Gamma_{m}^{\text{pc},x,i} & \text{if } i = \text{[throw X]} \\
\{ (\sigma_{m}^{\text{pc},i}, \cdot, \sigma_{m}^{\text{pc},i}) \} \cup H_{m}^{\text{pc},x,i} & \text{if } i = \text{Assertion} \\
\{ (\sigma_{m}^{\text{pc},i}, \cdot, \sigma_{m}^{\text{pc},i}) \} \cup H_{m}^{\text{pc},x,i} \cup \bigcup_{x \in MCA(a)} \{ (\sigma_{m}^{\text{pc},n}, \cdot, \sigma_{m}^{\text{pc},n}) \} & \text{if } i \in \text{MethodCall} \\
\bigcup_{n \in MCA(a)} \{ (\sigma_{m}^{\text{pc},n}, \cdot, \sigma_{m}^{\text{pc},n}) \} \cup \bigcup_{x \in MCA(a)} \{ (\sigma_{m}^{\text{pc},n}, \cdot, \sigma_{m}^{\text{pc},n}) \} & \text{if } i \in \text{MethodCall} \end{cases}
\end{align*}
receivers. A class $I^C_o[\sigma]$ refines $I^C_{o'}[\sigma]$ if it fulfills three requirements. First, the super class must be the same in both environments. The second requirement is that $I^C_o[\sigma]$ implements a subset of the interfaces implemented by $I^C_{o'}[\sigma]$. This is a relaxation from the definitions, and it is allowed because implementing fewer interfaces means that the original class is an over-approximation of the refined class. The last requirement is that the fields defined in the refined class is a superset of the fields in the original class. Intuitively, any analysis performed in the original classes accounted all the fields required by it. Thus, if the class will contain any new field afterwards, they were not referenced in the original class, and do not compromise the analysis.

The refinement between two methods $I^M_o[m]$ and $I^M_{o'}[m]$ is defined for three distinct cases. The first is between two methods with empty bodies. In this case, $I^M_o[m]$ is said to refine $I^M_{o'}[m]$ if the set of exceptions that the former guarantees not to propagate is a superset of the exceptions not propagated by the original method. That is, the refinement restricts the set of exceptions that a method may propagate. The second and simplest case, is the refinement between two concrete methods, i.e., without empty bodies: both $I^M_o[m]$ and $I^M_{o'}[m]$ must have the same code and handlers. The third, and most relevant case, is the one where an empty method $I^M_o[m]$ is refined by a concrete method $I^M_{o'}[m]$. Let EXCP be the auxiliary function which returns the set of exceptions that some method $I^M_o[m]$ propagates. The only constraint is that the refined method cannot propagate any of the exceptions declared to be caught and handled in the original method.
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 \( \Gamma \) be a closed environment, and \( \mathcal{G}_{MCA} \) be the instantiation of \( \mathcal{G} \) with MCA. Then \( \mathcal{G}_{MCA}(\Gamma) = \mathcal{G}(\Gamma) \).

**Proof.** The proof follows from the fact that if there are no missing components, then the function \( N^c_{pc} \) always falls into the first case. Thus, its definition is the same for both extraction algorithms. Also, the MCA algorithm outputs the same set of virtual method call receivers. Therefore, the extraction rules for \( \mathcal{G} \) are reduced to exactly the ones for \( \mathcal{G} \).

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

**Theorem 2.** Let \( \Gamma_o \) and \( \Gamma_o' \) be open environments, and \( m \) be a method signature implemented in both environments, i.e., \( \Gamma_o[m].code \neq \epsilon \) and \( \Gamma_o'[m].code \neq \epsilon \). Then \( \Gamma_o \preceq \Gamma_o' \) implies \( \Gamma_o \vdash \mathcal{G}m \subseteq \Gamma_o' \vdash \mathcal{G}m \).

**Proof.** The proof goes by case analysis on the BIR instructions set.

By the hypothesis, the method \( m \) must be implemented in both environments. Also by the hypothesis, \( \Gamma_o \preceq \Gamma_o' \) holds. Thus, from the refinement definition (Fig 16), the instructions array and exception table are the same. \( (\Gamma_o^{M}[m], code = \Gamma_o'^{M}[m], code \) and \( \Gamma_o^{M}[m], handle = \Gamma_o'^{M}[m], handle \).

It is trivial to see from the definition of function \( \mathcal{G} \) (Fig. 15 on page 22) that it outputs the same set of triples for all instruction groups whenever two methods have the same instructions array and exception table, except for \( NewObject \) and \( MethodCall \). Thus, the trivial cases are proven, and there are only two cases left.

The sets \( NewObject \) and \( MethodCall \) contain instructions that execute method invocations. The former contains the instructions that invoke object constructors. The later set contains the other method invocation instructions, either virtual or non-virtual. The non-virtual method calls, including constructor calls, have only one possible receiver. The possible receivers for a virtual method call, however, depend on the class hierarchy of each open environment. Moreover, the computation of \( N^c_{pc} \) also depends on the definitions of the open environment. Therefore, we present the proof for the virtual case for the \( MethodCall \). The other are special cases, where there is a single receiver for the call, and the proof is analogous.

The function \( \mathcal{G} \) produces a set of varying size for each possible receiver \( n \) of a virtual method call \( (n \in MCA(\mathcal{C},\mathcal{ns})) \): one normal edge, denoting a successful return from \( n \), plus pairs of edges for each exception that \( n \) may propagate (by function \( N^c_{pc} \)). First, we show that the set of receivers for a virtual method call in \( \Gamma_o[m].code \) is a subset of the receivers for the same call in \( \Gamma_o'[m].code \). Then, we conclude by showing that the set of propagated exceptions for a call to method \( n \) is always a subset in the refined environment.

Let \( MCA_{\Gamma_o'}(\mathcal{C},\mathcal{ns}) \) and \( MCA_{\Gamma_o}(\mathcal{C},\mathcal{ns}) \) be the sets of all possible receivers of the same virtual method call in the original and refined open environments,
respectively. They are defined as all the methods from the sub-type $C$ with the same signature $\text{ns}$. The static type $C$ can be either an interface, or a class. First, let $C$ be a class. The refinement relation defines that all classes in both $\Gamma_C^o$ and $\Gamma_C^o'$ have the same super class. The subtyping relation (Fig.6, page 9) defines that classes can only be subtyped by another class. Thus, the set of sub-types of $C$ is the same in both environments, and $\text{MCA}_{\Gamma^o} (C.\text{ns}) = \text{MCA}_{\Gamma'^o} (C.\text{ns})$.

Now let $C$ be an interface. Its sub-types are all the classes (and their sub-types) that implement the interface, or one of its sub-interfaces. The refinement relation defines that both the classes that implement an interface ($\Gamma_{\sigma}[\text{interfaces}]$ and its sub-interfaces ($\Gamma_{\omega}[\text{interfaces}]$), must be subsets in the refined open environment. Thus, the subtypes of $C$ in the refined environment are a subset of the original environment. Therefore, $\text{MCA}_{\Gamma'^o} (C.\text{ns}) \subseteq \text{MCA}_{\Gamma^o} (C.\text{ns})$.

Next, we show that the set of propagated exceptions by an arbitrary receiver of a method call in $\Gamma_M^o [m].\text{code}$ is a superset of the exceptions propagated by the same call in $\Gamma_M^o'[m].\text{code}$. Let $\Gamma_M^o'[n]$ be one possible receiver for a method invocation within $\Gamma_M^o [m]$. The refinement relation defines three cases for the method.

The first case is when both $\Gamma_o[n]$ and $\Gamma'_o[n]$ are empty methods. In this case, the set of exceptions guaranteed never to be propagated by the method in the refined environment must be a superset of the exceptions never propagated in the original version. The function $N_{\text{pc}}^e$ falls always into the second case, since the method implementation is missing in both open environments. The set of edges produced by this function is inversely monotone to the number of exceptions declared to be caught by the missing method ($E_{\Gamma_o - \Gamma_M^o'[n]}.\text{handlers}$). Therefore, the number of edges produced by $N_{\text{pc}}^e$ for the refined environment must be a subset of the edges produced for the original environment, since $\Gamma_o[n].\text{handlers} \supseteq \Gamma'_o[n].\text{handlers}$.

In the second case the method $\Gamma'_o[n]$ is implemented by $\Gamma_o[n]$. The refinement relation constrains the set of exceptions that the method implementation may propagate. It cannot contain an exception that the method declares not to propagate in the original environment. Therefore, if the refinement relation holds, the set of exceptions propagated by the refined method is clearly a subset of the exceptions propagated by the method from the original environment.

The third case is when $\Gamma_o[n]$ and $\Gamma'_o[n]$ are implemented methods. In this case, there is no constraint over the exceptions it may propagate. However, since both the code and handlers are preserved, the only possible difference in the CFG is in the case that a method called within $\Gamma_o[n]$ has been implemented, or is still a missing method. Thus, by the two previous cases, the set of propagated exceptions in the refined environment still has to be a subset of the original environment.

We conclude that the set of possible receivers for a virtual method call, and the set of propagated exceptions for a method invocation, are both a subset in the refined environment, when compared to the original environment. Therefore, the set of triples is also a subset.
The above results ensure soundness 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 [3, 12]. 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 ([12, Th. 36]). Third, temporal safety properties are preserved (backwards) under behavioral simulation ([12, Cor. 17]). These three results guarantee preservation of temporal safety properties under refinement of open environments. Together with the soundness result for $G$ established in [2] and Theorem 1 above, we obtain soundness of $oG$.

Thus, 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 negatives, intrinsic to sound over-approximation, can thus be alleviated through CFG re-extraction. As pointed out in the proof sketch, 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 instantiation of new code produces a refined model due to the fewer over-approximations w.r.t. exceptional flow.

8 The ConFlEx Tool

The algorithms presented in Sections 6 and 7 have been implemented in the ConFlEx tool. In the following sections we describe the implementation aspects, and present practical results.

8.1 Implementation

This section describes the Control Flow graph Extractor tool (ConFlEx) [5], which implements the two CFG extraction algorithms: $G$ (Def. 9), and $oG$ (Def. 11).

The tool is written in OCaml, and uses SawJa [4, 13], a library for the static analysis of Java bytecode programs. SawJa provides high-level functions to manipulate bytecode .class files, implements algorithms for virtual method call resolution, and transformations from bytecode into intermediate representations.

We have tailored SawJa in several parts. The major enhancement was on the BC2BIR transformation, to provide an accurate estimation of the possible exception types raised by the BIR instruction [throw]. In the standard implementation of SawJa, the BC2BIR transformation does not consider the program’s class hierarchy. It only performs a syntactic transformation, and associates java.lang.Object to expressions and variables of non-primitive types. We altered the symbolic execution of BC2BIR to associate types to variables and expressions. Also, in operations involving non-primitive types, we compute the
type as the common super-type between the operands. This is a conservative estimation of the actual type, but still sound for modular set-ups.

We have implemented the formal definitions that we have introduced as new modules in Sawja. The module DefCFG represents and manipulates CFGs, as presented in the Definitions 4 and 6. We wrote it as a separate module to make ConFLEx independent from a single CFG definition. This facilitates in the future to add other CFGs definitions. The OpenEnv module implements the representation of an open Java bytecode system as an open environment, following the definitions in Figure 14. The environment is represented by a structure containing three associative maps, representing $\Gamma^I_o$, $\Gamma^C_o$, and $\Gamma^M_o$. The module also implements the check of the refinement relation, following the rules in Figure 10. The module MCA implements the virtual method call resolution algorithm for open systems. The algorithm over-approximates the set of possible receivers to a virtual call as presented in Section 7.1: the set of all methods with same signature, and from a subclass, of the invocation instruction’s operand. We have adapted the code of the Class Reachability Analysis (CRA), native from Sawja, to implement MCA.

The interfaces for the missing components are provided as a combination of Java annotations [27], and dummy methods containing a single return instruction. The use of Java annotations allows us to set the granularity of components to method-level. Moreover, Sawja conveniently provides built-in support for the manipulation of Java annotations. The dummy methods are necessary because the annotations must be associated to a method in the .class file.

```java
import java.lang.annotation.*;

@Retention(value = RetentionPolicy.CLASS)
@Target(value = {ElementType.CONSTRUCTOR,ElementType.METHOD})
public @interface GhostComponent
{
    String[] req_meths();
    String[] handlers();
}
```

Fig. 17: The GhostComponent annotation

We have defined the GhostComponent annotation template, containing the mandatory annotations for the missing methods. The user must compile a .class file with the code in Figure 17 and put it into each directory containing an annotated method. The user annotates a method by adding a custom GhostComponent annotation immediately before the declaration of the dummy method. The field req_meths is an array of strings specifying the methods declared to be invoked in the method code. The field handlers is an array of strings specifying which exceptions the user declares that the missing method will never propagate when
its code becomes available. The standard built-in annotations \texttt{Retention} and \texttt{Target} defines the levels of visibility and granularity of the annotation, respectively. The former states that the annotation is only visible in the class file, but not at run-time; this suffices for our purposes. The latter determines that the elements which can be annotated are methods, both standard and constructor.

\textit{Example 6 (Annotated missing method).} Figure 18 shows the annotation of the method \texttt{even} reflecting its interface, as defined in Example 4.

\begin{verbatim}
@GhostComponent(
    req_meths = { "boolean odd(int)" },
    handlers = { }
)
public boolean even(int a) {
    return a;
}
\end{verbatim}

Fig. 18: Example of an annotated dummy method

By default, \textsc{ConFlEx} does not consider methods from the standard Java library (API) to be part of the program, and it does not extract their CFGs. The tool considers only the client program, which is the code that the user explicitly declares to belong to the program. The client code must contain an entry method \texttt{main}. \textsc{ConFlEx} assumes that calls to methods from the API are side-effect free. That is, there cannot be call-backs, which are calls within the API methods to method from the client program. Also, we assume that API methods can only propagate exceptions declared in the \texttt{throws} field of the method’s signature. The user can alter \textsc{ConFlEx}'s assumptions w.r.t to API methods by explicitly declaring the standard API to be a component of the program. Unfortunately the Java standard API is large, and the consequence are unnecessarily large CFGs. That is the reason behind the design choice to extract only the client code by default.

We have introduced in Section 7.1 the set \( E_{\Gamma} \) of all exception types in an open environment, and have presented it in Definition 10 as all the subtypes of \texttt{java.lang.Throwable}. In practice, we can filter out exception types which can not be raised because they are not referenced neither in the code, nor in the interfaces. We compute the \( E_{\Gamma} \) as the union of: (a) exceptions represented by the BIR assertions (Fig. 8); (b) user-defined exception classes; and (c) exceptions declared as potentially throwable by the API methods.

\textsc{ConFlEx} implements the caching of the intra-procedural analysis. The caching has two benefits. First, it allows to check if the refinement relation holds between two versions of the program. Second, it allows the incremental extraction of newly provided components, in contrast to an entire new intra-procedural analysis. This is valid even if an open environment is not a refinement of another,
as long as ConFlEx is executed with the same configuration options. The feature exploits the fact that the computation of the intra-analysis is preserved if the implementation of a method is not altered. Still, ConFlEx recomputes the entire inter-procedural analysis, so that the control flow caused by propagated exceptions is over-approximated the least as possible.

ConFlEx creates two files for caching of previous analyses, all with the same name: a text file, and an XML file. The text file contains the options used in the tool execution. This is required to check whether two versions of the program have been analyzed in the same set-up. The XML file stores the results from the intra-procedural analysis: the CFG, the set of propagated exceptions, and method calling points. ConFlEx uses the external library XML-Light [26] for parsing the XML files.

8.2 Experimental Results

We validate our tool by using real-world Java applications. We emulate open Java bytecode systems by taking a complete JBC applications, replacing the implementation of some of the classes with annotated methods. Then, we re-introduce the implementations incrementally, to mimic the arrival of code.

We have considered the scenarios presented in Table 1. They reflect the incremental refinements we have performed on each program. Specifically, by performing the first refinement over the initial open system (Scenario 1), we have the Scenarios 2 and 3. Also, the Scenario 4 represents the implementation of the resulting open system from Scenario 3 into a closed environment. There is no refinement in Scenarios 5 and 6, and the extraction of CFGs is performed over the original closed programs.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>System Type</th>
<th># missing classes</th>
<th>VMC algorithm</th>
<th>Reuse Intra</th>
<th>Refines Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open</td>
<td>4</td>
<td>MCA</td>
<td>No</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>Open</td>
<td>1</td>
<td>MCA</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Open</td>
<td>1</td>
<td>MCA</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Closed</td>
<td>0</td>
<td>MCA</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Closed</td>
<td>0</td>
<td>MCA</td>
<td>no</td>
<td>×</td>
</tr>
<tr>
<td>6</td>
<td>Closed</td>
<td>0</td>
<td>RTA</td>
<td>No</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 1: Extraction scenarios

The three programs considered were Jasmin version 2.4 [25]; JavaCUP version 11a beta [13]; and JFlex version 1.4.3 [21]. We have artificially generated the missing methods by replacing the actual implementations with dummy methods, and GhostComponent annotations. The req_meths field was annotated with the methods that are called in the actual method implementation. The handler field was empty in all cases. That is, we considered that the missing methods could potentially propagate any exception in the $E_{\Gamma_1}$ set. The missing classes in Scenario
are: ClassFile, InsnInfo, Scanner and parser for Jasmin; NFA, SemCheck, RegExp and IntCharSet for JFlex; symbol_set, terminal_set, lalr_state and production for JavaCUP. The missing classes in Scenarios 2 and 3 are: Scanner for Jasmin; NFA for JFlex; production for JavaCUP.

Table 2 shows the experimental results. All tests have been made on an Intel i3 2.27 GHz with 4GB of RAM. The considered data are: number of JBC and BIR instructions; number of nodes and edges of the CFG after the inter-procedural analysis; the parsing time; time of the intra-procedural analysis; and time for the inter-procedural analysis.

We can draw several conclusions from the experimental results. First, the number of BIR instructions is less than 40% of bytecode instructions, for all cases. This indicates that the use of BIR alleviates the blow-up of CFGs, and clearly the program analysis benefits from this.

We observe that the number of unavailable components has a significant impact on the size of the over-approximations. For instance, step 1 where four classes are missing and thus has fewer instructions, produces larger CFGs than steps 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>
<thead>
<tr>
<th>Step</th>
<th>VMC</th>
<th>Reused results</th>
<th>Missing classes</th>
<th># of JBC instructions</th>
<th># of Nodes</th>
<th># of Edges</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Intra</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inter</td>
</tr>
<tr>
<td>1</td>
<td>MCA</td>
<td>no</td>
<td>4</td>
<td>25440</td>
<td>53467</td>
<td>54285</td>
<td>1256</td>
</tr>
<tr>
<td>2</td>
<td>MCA</td>
<td>yes</td>
<td>1</td>
<td>30377</td>
<td>35684</td>
<td>36228</td>
<td>291</td>
</tr>
<tr>
<td>3</td>
<td>MCA</td>
<td>no</td>
<td>1</td>
<td>30377</td>
<td>35684</td>
<td>36228</td>
<td>1540</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>yes</td>
<td>0</td>
<td>32223</td>
<td>34411</td>
<td>35052</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>no</td>
<td>0</td>
<td>32223</td>
<td>34411</td>
<td>35052</td>
<td>1554</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>yes</td>
<td>0</td>
<td>30530</td>
<td>27267</td>
<td>27717</td>
<td>690</td>
</tr>
<tr>
<td>7</td>
<td>MCA</td>
<td>no</td>
<td>0</td>
<td>30930</td>
<td>27267</td>
<td>27717</td>
<td>690</td>
</tr>
<tr>
<td>8</td>
<td>MCA</td>
<td>yes</td>
<td>1</td>
<td>33354</td>
<td>76798</td>
<td>77649</td>
<td>567</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>no</td>
<td>1</td>
<td>33354</td>
<td>76798</td>
<td>77649</td>
<td>2098</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>yes</td>
<td>0</td>
<td>35422</td>
<td>45455</td>
<td>46328</td>
<td>66</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>no</td>
<td>0</td>
<td>35422</td>
<td>45455</td>
<td>46328</td>
<td>2126</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>yes</td>
<td>0</td>
<td>32049</td>
<td>32097</td>
<td>32509</td>
<td>983</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>no</td>
<td>0</td>
<td>53474</td>
<td>53856</td>
<td>54777</td>
<td>1676</td>
</tr>
</tbody>
</table>
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 (step 3) produces 43% more nodes as compared to RTA (step 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 lightweight 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 lightweight, such as in the ATM example mentioned in Section 2.

We do not provide comparative data with other extraction tools, such as Soot [22] or Wala [17] because this would demand the implementation of similar extraction rules from their intermediate representations. However, experimental results from Sawja [13] 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.

9 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 [6], 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 [22], 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 [7] 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 [17], 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 [20] 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. [23] 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.

10 Conclusion

The present work presents a modular framework for the extraction of sound control-flow graphs from open Java bytecode systems, considering precise exceptional control-flow. Despite an open Java bytecode program cannot execute, still one may analyze the available components. The extracted CFGs from both algorithms are suitable for various control-flow analyses, in special for formal methods. The compositional verification principle proposed by Gurov et al. [12], is an example of a technique that directly benefits from the modular framework.

We based our work on two previous results. First, we generalized the definition of the Java Virtual Machine framework from [10] to formally define open systems. On it, the missing components are represented by user-provided interfaces. Moreover, we generalized a previous CFG algorithm, originally defined for closed system, to extract CFGs from the available components of an open Java bytecode system. The inter-dependencies involving missing components are resolved using the information from the interfaces. We define a refinement relation, which are constraints over the arrival of missing components, and prove that if the relation holds, then the CFG extracted previously are still sound. Therefore, safety properties verified over such CFGs hold for any closed system that implements the original open system.

The framework is implemented in the CONFlEx tool [5]. It uses Sawja, a library for the static analysis of Java bytecode, for the BIR transformation,
and virtual method call resolution. We have tailored Sawja in several aspects: improved the analysis of explicit exceptions type, adapted one of its VMC algorithms to implement our sound modular VMC resolution, implemented the representation of an open program, and the refinement relation. The ConFLEx tool implements the extraction algorithm, and the CFG representation. Also, we have added the analysis reusage feature, to enable the incremental CFG extraction.

The experimental results have confirmed that the over-approximations necessary to make the CFGs sound have significantly impacted the size of the extracted control-flow graphs. Unfortunately they are necessary to achieve sound results. However, the modular algorithms had similar efficiency compared to the previous algorithm for closed systems. Both were linear w.r.t the number of instructions. At last, the reuse of previous results, and consequent incremental extraction, has improved dramatically the efficiency of the extraction of open systems, when compared to an entire new extraction.

10.1 Future Work

The proposed method performs excessive over-approximations, necessary to achieve soundness. As it is, the approach we presented provides possibilities for further developments and refinements. From a theoretical point of view, future efforts might be put in refining some of the concerned issues, e.g. the definition of the set of exceptions possibly propagated by a method whose instructions are not available. The idea is to work on these issues to be restrictive as little as possible when defining the refinement rules for an open environment, and at the same time to over-approximate as little as possible when computing some specific set of data.

About the implementation, some other solutions can be proposed for tuning performance, storing and presenting the analysis results in the output. A possible technique to speed up the tool is to parallelize the extraction step of the algorithm when executing the intra-procedural step, by using a multi-threading strategy. Each thread would analyze either one single method or all the methods of a single class, as the number of methods considered and related threads created in the former suggestion might be too large. The partial results related to each class would be merged at the end of the intra-procedural analysis, before proceeding with the inter-procedural phase. A different solution about the storage and loading of data may be to use Database structures instead of text files. In this way the time needed to perform the storing and the loading steps could improve significantly.

References


18. OpenBravo Inc.: OpenBravo site (2013), \url{http://www.openbravo.com/}

A Open Environment Storing File Definition

The open environment representing a given program is stored in an XML file, as described in Section 8.1. The XML file also stores some other partial results obtained during an execution of the tool. The data are stored following the structure defined in a separate DTD file, shown in Figure 19. The content of the XML file is validated against the DTD format during the loading step.
Fig. 19: DTD file stored.