Automatic Extraction of Program Models for Formal Software Verification

PEDRO DE CARVALHO GOMES

Doctoral Thesis in Computer Science
Stockholm, Sweden, 2015
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi fredagen den 4 december 2015 klockan 14.00 i sal F3, Kungliga Tekniska högskolan, Lindstedtsvägen 26, Stockholm.

© Pedro de Carvalho Gomes, October 30, 2015

Tryck: US-AB
Abstract

In this thesis we present a study of the generation of abstract program models from programs in real-world programming languages that are employed in the formal verification of software. The thesis is divided into three parts, which cover distinct types of software systems, programming languages, verification scenarios, program models and properties.

The first part presents an algorithm for the extraction of control flow graphs from sequential Java bytecode programs. The graphs are tailored for a compositional technique for the verification of temporal control flow safety properties. We prove that the extracted models soundly over-approximate the program behaviour w.r.t. sequences of method invocations and exceptions. Therefore, the properties that are established with the compositional technique over the control flow graphs also hold for the programs. We implement the algorithm as ConFlEx, and evaluate the tool on a number of test cases.

The second part presents a technique to generate program models from incomplete software systems, i.e., programs where the implementation of at least one of the components is not available. We first define a framework to represent incomplete Java bytecode programs, and extend the algorithm presented in the first part to handle missing code. Then, we introduce refinement rules, i.e., conditions for instantiating the missing code, and prove that the rules preserve properties established over control flow graphs extracted from incomplete programs. We have extended ConFlEx to support the new definitions, and re-evaluate the tool, now over test cases of incomplete programs.

The third part addresses the verification of multithreaded programs. We present a technique to prove the following property of synchronization with condition variables: “If every thread synchronizing under the same condition eventually enters its synchronization block, then every thread will eventually exit the synchronization”. To support the verification, we first propose SyncTask, a simple intermediate language for specifying synchronized parallel computations. Then, we propose an annotation language for Java programs to assist the automatic extraction of SyncTask programs, and show that, for correctly annotated programs, the above-mentioned property holds if and only if the corresponding SyncTask program terminates. We reduce the termination problem into a reachability problem on Coloured Petri Nets. We define an algorithm to extract nets from SyncTask programs, and show that a program terminates if and only if its corresponding net always reaches a particular set of dead configurations. The extraction of SyncTask programs and their translation into Petri nets is implemented as the STaVe tool. We evaluate the technique by feeding annotated Java programs to STaVe, then verifying the extracted nets with a standard Coloured Petri Net analysis tool.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Contribution</td>
<td>8</td>
</tr>
<tr>
<td>1.2 Organization</td>
<td>12</td>
</tr>
<tr>
<td>2 Background</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Java Bytecode and the Java Virtual Machine</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Control Flow Graphs</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Direct Extraction of CFGs from Bytecode</td>
<td>23</td>
</tr>
<tr>
<td>2.4 The BIR Language</td>
<td>27</td>
</tr>
<tr>
<td>2.5 Compositional Verification</td>
<td>30</td>
</tr>
<tr>
<td>2.6 Weak Simulation on Models</td>
<td>34</td>
</tr>
<tr>
<td>2.7 Hierarchical Coloured Petri Nets</td>
<td>35</td>
</tr>
<tr>
<td>3 CFG Extraction from Complete Programs</td>
<td>39</td>
</tr>
<tr>
<td>3.1 Extraction of CFGs from BIR</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Correctness of CFG Extraction</td>
<td>42</td>
</tr>
<tr>
<td>3.3 The ConFlEx Tool for Complete Programs</td>
<td>46</td>
</tr>
<tr>
<td>3.4 Discussion</td>
<td>48</td>
</tr>
<tr>
<td>3.5 Related Work</td>
<td>50</td>
</tr>
<tr>
<td>4 CFG Extraction from Incomplete Programs</td>
<td>53</td>
</tr>
<tr>
<td>4.1 Incomplete Java bytecode programs</td>
<td>53</td>
</tr>
<tr>
<td>4.2 The 65 Extraction Algorithm</td>
<td>55</td>
</tr>
<tr>
<td>4.3 Correctness Proof</td>
<td>57</td>
</tr>
<tr>
<td>4.4 The ConFlEx Tool for Incomplete programs</td>
<td>59</td>
</tr>
<tr>
<td>4.5 Related Work</td>
<td>64</td>
</tr>
<tr>
<td>5 Verifying Synchronization with Condition Variables</td>
<td>67</td>
</tr>
<tr>
<td>5.1 Overview of the Approach</td>
<td>68</td>
</tr>
</tbody>
</table>
Acknowledgements

It has been more than five years since I swapped the heat, family and friends in Belo Horizonte for a promising but uncertain journey to Stockholm. Looking back now, I am sure that this was the best choice I could have ever made. I have met so many people that helped me so much, and I will try to be fair to you all. However, you have been so many that I apologize beforehand if I forget to mention someone.

Dilian Gurov: you have been an insightful and committed supervisor. Every single discussion that we had has been rewarding, and I have learned from you more than I can say in few word. Thank you very much for all the support. Marieke Huisman: thank you very much for the cooperation and hard work. Afshin Amighi: this thesis is a continuation of your brilliant work. Thank you for the cooperation. Attilio Picoco: you were a great research partner. I wish you all the best in your career. Sérgio Campos and Alex Borges: thank you once again for the collaboration, despite the distance. Siavash Soleimanifard: it was great working with you. I am sure there is nothing but success in your way. Karl Palmskog and Björn Terelius: thank you for the nice discussions along the PhD. Musard Balliu: I had lots of nice discussions and fun working with you. Thank you very much for the support. Susanna Rezende and Jan Elffers: It has been a pleasure sharing office with you. Philip Haller: thank you for patiently reviewing the current thesis.

To the people from the Theoretical Computer Science department: Adam Collberg, Andreas Lundblad, Benjamin Greschbach, Cenny Wenner, Christoph Baumann, Emma Enström, Guillermo Rodríguez, Gunnar Kreitz, Hamed Nemati, Jakob Nordström, Jana Götte, Johan Håstad, Karl Meinke, Linda Kann, Lukáš Poláček, Mads Dam, Marc Vinyals, Mladen Mikša, Oleksandr Bodriagov, Oliver Schwarz, Roberto Guanciale, Sangxia Huang, Sonja Buchegger, Stefan Nilsson, Tobiörn Granlund, Vahid Mosavat, and others. You have created a great research environment. Thank you all for the support, and for the fun!

To the friends in Sweden: Alexandre Sandim, Ankur Gupta, Daniel Albernaz, Dany Toumie, David Stoltz, Emma Lyngedal, Fabio Bittencourt, Felipe Oliveira, Flávio Mazzaro, Gustavo Baldissera, Henrik Holst, Iris Engström, Jenny Morales, Johan Hahn, Rafael Barros, Raquel Bohn, Rodrigo Vilela, Sergey Popov, Siri Gilllund, Susanne Froröth, Tiago Froés, Usman Rafique, Vesela Magyarova, and all the others: thank you a lot for making me feel like home, even in such cold weather. A special salute for the Esporte Clube Pau Grande Estocolmo: Vai Pauzão!
ACKNOWLEDGEMENTS

To all friends in Brazil, and in other countries as well. Thank you very much for the kind support. I am especially grateful to all the ones that could visit me in Stockholm. Each visit brought lots of joy, and confirmed that friendship does not know distances.

I am grateful to my dear homeland Brazil, and all our people. It was just because I had the privilege to study in a high-level public university that I am here today. Nevermind about football. We have way more to offer, though we do not always realize it. Let us never loose our resilience and hope for better days. I also express my gratitude to Sweden for the given opportunity, and for being such a welcoming home.

To conclude, this thesis is dedicated to my family: Elena (and family), Lucas (and family), mom (Patricinha), and dad (Paulo). Having you make me the happy person that I am. I also dedicate the thesis to my dear Hilda. Thank you for the patience, love, and for being there.
Chapter 1

Introduction

Software systems are omnipresent in contemporary society. They are employed in virtually any area that affects people’s life, and unexpected errors may lead to undesirable time and money losses, and in the extreme case, even jeopardize lives. We have seen in the last decades an increasing demand for software quality and reliability, which has been followed by steadily increasing investments in software development processes. For example, software testing has gained a lot of attention, and has helped to improve the overall quality of software. Unfortunately, critical flaws are still being uncovered, causing harm to businesses, and to society in general. It is clear that the available techniques have not met the expectations for software quality. Thus, it is crucial to develop new techniques that can assure a higher level of software trustworthiness.

In this context, mathematically based formal verification techniques have gained growing acceptance as a means to ensure the reliability of software systems. Different formal techniques have been developed to address this goal, such as various static analyses, model checking and (automated) theorem proving. In contrast to testing methods, which uncover as many bugs as possible, but cannot guarantee their absence, formal verification techniques are exhaustive with respect to the property being verified. Unfortunately, the major obstacle for formal verification techniques is the state space of software, which is typically very large or even infinite. As program sizes increase, the combinatorial explosion of the state space limits the range of properties that one can verify, and it even makes the verification intractable in most cases.

Among the approaches to circumvent the problem, one of the most popular is to verify properties over an abstract program model, i.e., a model representing the program which only preserves the relevant information to establish the property at hand. Ideally, a program model is created by an algorithm that: takes into account all features of a programming language; is formally defined and followed by a correctness argument that establishes which class of properties the extracted models preserve; and is implemented as an automated tool. However, given the
complex semantics of popular programming languages, the hardness to formally establish correctness results, and the pressure for software releases, it is rarely the case that extraction algorithms comply to all the desired requirements. As a consequence, there is no guarantee that the properties being verified are preserved by the abstract program models, and thus the verification results cannot be trusted.

In this thesis we propose different techniques for the automatic extraction of abstract program models, focusing on their application in formal verification. The thesis is divided into three major parts, which encompass diverse aspects of program analysis and formal verification. For instance, the presented techniques cover both the verification of safety and liveness program properties, sequential and concurrent programs, complete and incomplete software systems, source and executable languages. Moreover, the techniques address challenging constructs for the analysis of programs in fully-fledged popular languages, such as dynamic dispatching (i.e., virtual methods), exceptions, and synchronization primitives. In all cases, we highlight the formal aspects of our proposed techniques.

The first part presents an algorithm for the extraction of control flow graphs (CFGs) \[2\] from sequential Java bytecode (JBC) programs, which is sound w.r.t. sequences of method invocations and exceptions. CFGs are a common abstract program model where the control flow information is kept, and all program data is abstracted away (see e.g. \[12, 71\]). In a CFG, nodes represent the control points of the program, and the edges represent the move of control between control points.

The extracted CFGs are tailored for compositional verification of control flow based temporal safety properties in the style of \[35, 39\]. Our definition of CFGs makes two adaptations to the standard notion, to suit both the compositional technique, and the verification of JBC. First, there are no explicit inter-procedural edges: method calls are represented by labels on the outgoing edges of invocation nodes, and return points are depicted as atomic propositions on sink nodes. Second, the CFGs contain exceptional control nodes, i.e., nodes representing the takeover of control by the JVM to handle the exception.

The algorithm is defined for Java bytecode, which is the executable language of the Java Virtual Machine (JVM). We focus on JBC because we want to extract CFGs even in the absence of the source code. For example, one may want to verify a system in which one of the components is provided by a third-party, but only as target code. In addition, we avoid any possible compiler-related issues, and we can analyze code written in other programming languages than Java that also compiles to JBC, such as Scala \[63\].

The control flow analysis, when considering exceptions, is challenging for two main reasons. First, the stack-based nature of the JVM makes it hard to determine the type of explicitly thrown exceptions, thus making it difficult to statically decide to which handler (if any) control will be transferred. Second, the JVM can raise (implicit) run-time exceptions, such as NullPointerException and IndexOutOfBoundsException.
BoundsException, by the abnormal execution of some of its instructions. Keeping track of where such exceptions can be raised requires much care. Also, if a method does not handle an exception, its execution is aborted, and the exception is propagated to its caller method. The computation of control flow caused by exception propagation is not trivial because of the inter-dependencies between the program’s methods. Similar works about control flow analysis have neglected the exceptional control flow because of the complexity it adds [79, 47].

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

Numerous approaches to the automatic extraction of control flow graphs from program code have previously been presented (see, e.g., [46,16,47]). However, these are typically not accompanied by any formal correctness argument. We attempt to fill this gap: we present a CFG extraction algorithm for sequential (i.e., single-threaded) Java bytecode that captures normal as well as exceptional control flow, and we prove that the extraction algorithm is sound w.r.t. program behaviour, if the latter is viewed as a set of sequential executions (runs). The main challenge here is to come up with a simple formalization of the extraction that allows a relatively straightforward (even if large) soundness proof, to pave the ground for fully formal soundness proofs by means of theorem provers. One complication here derives from the fact that the notion of correctness of the extraction algorithm is indirect, in terms of the extracted CFGs being sound models w.r.t. the programs from which they are extracted.

The starting point of our algorithm is an idealized CFG extraction algorithm proposed by Amighi [3], which is proven to simulate the JVM behaviour w.r.t. sequences of method invocations and exceptions. The algorithm follows the philosophy of Freund and Mitchell underlying their formalization of the JVM to abstract from the complications arising from exceptional flows and to relativize the extraction on an oracle that is able to look into the stack and predict the exceptions that can be raised at each instruction [29]. The resulting, conceptually simple, idealized algorithm serves as a specification for concrete CFG extraction algorithms, which have to implement the oracle in a suitable fashion. The CFGs extracted by the algorithm, however, are rather verbose: in bytecode, all operands are on the stack, thus many instructions for stack manipulation are present, all giving rise to irrelevant edges in the CFG. This affects negatively the efficiency of verification of control flow properties.

Our CFG extraction algorithm implements the oracle and at the same time produces more compact CFGs. The algorithm consists of two separate transformations. The first one converts the JBC program into an intermediate representation as a
BIR program. The second transformation defines CFG extraction from BIR. BIR is a stack-less representation of Java bytecode developed by Demange et al. [26]. Thus all instructions (including the explicit athrow) are directly connected to their operands, providing the necessary information to implement the oracle. Also, the BIR transformation inserts assertions along the program representation, denoting that a run-time exception can be raised at a given program point. Further, the representation of a program in BIR is smaller than in JBC, because operations are not stack-based, but represented as expression trees. As a result, the extracted CFGs are more compact.

The composition of the two transformations constitutes the concrete CFG extraction algorithm from JBC. Its correctness proof uses the correctness of the idealized algorithm. We prove that the CFGs extracted by the idealized algorithm are simulated structurally (rather than behaviourally) by the CFGs extracted by the concrete algorithm, which significantly simplifies and shortens the proof. By reusing a previous result from [39] that structural simulation induces behavioural simulation, and by transitivity of simulation, we can deduce behavioural simulation.

The concrete algorithm is implemented as the tool ConFlEx. It uses SawJa [37], a library for static analysis of Java bytecode, for the virtual method resolution, and for the BIR transformation. The BIR transformation in SawJa is purely syntactic. Therefore, we instrumented it to associate types to operands, and to compute the most general type when some operation is performed over operands of different types. Currently, SawJa provides assertions for a subset of the run-time exceptions. ConFlEx supports all the available assertions, and can easily be extended if more are provided. Next, we implemented the CFG extraction algorithm from the BIR representation. It is subdivided into two distinct analyses. The first is intra-procedural, and the CFG of a method is extracted by analyzing its instructions only. The second analysis is inter-procedural, and ConFlEx uses a fixed-point computation to determine the flow caused by propagation of uncaught exceptions.

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

The second part presents a framework for the extraction of CFGs from the available components of incomplete Java bytecode programs.

We define incomplete programs as those where the implementation of some components is not yet available. Typical situations when one has to deal with such programs are systems depending on mobile code, or systems under development. Despite the absence of code, it is still possible to establish properties of the available
code, and (under some assumptions) even global program properties, with formal techniques such as the compositional method mentioned above. Previous techniques have been proposed to analyze incomplete JBC programs \cite{19, 54}; 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 and exceptions impose additional difficulties in a scenario where components are not available. E.g., a potential raise of an exception is directly related to the implementation of a software component, and few facts can be inferred in the absence of code. 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 first proposing a formal modelling of incomplete Java bytecode programs. 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.

Next, we generalize the algorithm for complete JBC programs defined in Part I. 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 behaviour (as defined by the JVM), as long as the specified constraints are respected. Therefore, already established behavioural or structural properties are thus guaranteed to still hold.

We have extended the ConFlEx tool to support the extraction of CFGs from incomplete programs. It features caching of previous analyses, necessary for the incremental refinement, and matching of newly arriving code against their interface specifications. Originally, SAWJ could not analyze incomplete programs. We have extended it to support incomplete Java bytecode systems. This includes the implementation of a sound VMC resolution algorithm for modular set-ups, data structures to represent interfaces for the unavailable code, and the verification of
the refinement relation.

We have re-evaluated ConFLEx with a number of experiments. Our test cases mimic incomplete JBC programs by taking a complete program, replacing some components with interfaces, and incrementally re-instantiating the removed code. We have chosen this methodology to have a quantitative picture of how much the over-approximations for incomplete programs impact the CFGs size and extraction time when compared to extracting CFGs from complete programs. 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.

The third part presents a technique for verifying the synchronization of multithreaded programs with condition variables (CVs) by means of an intermediate language, and Coloured Petri Nets (CPN).

CVs are a synchronization mechanism to coordinate multithreaded programs, used in conjunction with locks. Threads can wait on a CV, meaning they suspend their execution until another thread notifies the CV, causing the waiting threads to resume their execution. The signaling is asynchronous: if no thread was waiting on the CV, then the notification has no effect. CVs are used in conjunction with locks; a thread must acquire the associated lock for notifying or waiting on a CV, and if notified, must reacquire the lock.

Many widely used programming languages feature condition variables. In Java, for instance, condition variables are provided both natively as an object’s monitor, i.e., a pair of a lock and a CV, and in the concurrent API, as one-to-many Condition objects associated to a Lock object. Nevertheless, condition variables have not been addressed sufficiently with formal techniques, mainly because of the complexity of reasoning about asynchronous signaling. For instance, Leino et al. [56] acknowledge that verifying the absence of deadlocks when using condition variables is hard because a notification is “lost” if no thread is waiting on it. Thus, one cannot reason locally whether a waiting thread will eventually be notified. The correct usage of CVs involves both control flow and data flow aspects, and directly depends on the global thread composition, i.e., the type and quantity of threads executing in parallel.

The synchronization property of interest is the following: “If for every set of condition variables, every thread synchronizing under the variables of the set eventually enters its synchronization block, then every thread will eventually exit the synchronization block”. The property, here stated informally, entails that no thread will block indefinitely because of erroneous synchronization. E.g., no thread will wait indefinitely for the notification of another thread. To the best of our knowledge, the present work is the first to address a liveness property involving condition variables. As the verification of such properties is undecidable in general, to stay within a decidable fragment, we limit our technique to programs with bounded data domains and numbers of threads. Still, the verification problem is subject to a
combinatorial explosion of thread interleavings. Our technique alleviates the state space explosion by isolating the relevant aspects of the synchronization.

First, we study the liveness property in the context of a synchronization specification language. To this end, we introduce \textit{SyncTask}, a simple concurrent language where all computations occur inside synchronized code blocks. It has been designed to capture common patterns of CV usage, while abstracting from all irrelevant details. SyncTask is a programming language-independent, intermediate representation of finite synchronization schemes, for which the verification is decidable. It has a Java-like syntax and semantics, and features the relevant constructs for synchronization, such as locks, CVs, conditional statements, and arithmetic operations. However, it is non-procedural, data types are bounded, and it does not allow dynamic thread creation. These restrictions render the state space of SyncTask programs finite, and make the termination problem decidable.

We transform the termination problem of SyncTask programs into a reachability problem on hierarchical Coloured Petri Nets \cite{42}. CPNs provide a suitable balance between expressiveness and analysability, and allow a concise modelling of the control flow of multi-threaded programs. CPNs have been used successfully over the last decades for modelling concurrent systems, and are supported by mature analysis tools such as CPN Tools \cite{44}. We model the constructs of SyncTask as CPN components, and describe how to extract CPNs automatically from SyncTask programs. Then, we establish that a SyncTask program terminates if and only if the extracted CPN always reaches dead configurations (i.e., configurations without successors) where the tokens representing the threads are in a unique \textit{end place}.

The termination condition can be verified algorithmically with the computation of the reachability graph of Petri Nets, and the check that: (i) there are no cycles in the graph (meaning unconditional termination), and (ii) the only dead configurations are those where the end place contains all thread tokens. Standard CPN analysis tools can efficiently compute the reachability graphs, and the complexity of these checks is linear in the size of the graph. Also, in case that the condition does not hold, an inspection of the reachability graph easily provides the cause of the non-termination.

Next, we address the problem of verifying the correct usage of CVs in real concurrent programming languages by showing how to verify the synchronization in Java programs, if these are bounded. There is a consensus in Software Engineering that the synchronization schemes must be defined as minimal as possible, both to minimize the risk of error conditions and to avoid the latency of blocking threads. As a consequence, many programs present a finite (though arbitrarily large) synchronization behaviour. The analysis of synchronization in Java programs is undecidable in general. We therefore introduce an annotation scheme to assist the automatic extraction of SyncTask programs. For instance, the user must annotate the creation of threads, and provide the initial state of the variables accessed inside the synchronized blocks. We establish that for correctly annotated, bounded Java programs, the synchronization property discussed above is equivalent to termination of the extracted SyncTask program.
The extraction of SyncTask programs from annotated Java programs, and the translation of SyncTask programs into CPNs is implemented as the STaVe tool. We validate the verification technique on two test cases by generating CPNs from annotated Java programs and analyzing these with CPN Tools. The first test case evaluates the scalability of the tool w.r.t. the number of synchronizing threads. It is an implementation of a shared buffer, for which we performed experiments with different numbers of threads and buffer sizes. The results show the expected exponential blow-up of the state space, but we were still able to analyze the synchronization of several dozens of threads. The second test case evaluates the scalability of the tool w.r.t. the size of program code that does not affect the synchronization behaviour of the program. For this we annotated the Java source code of PIPE [27], another Coloured Petri Nets analysis tool that is large, but exhibits a simple synchronization behaviour.

1.1 Contribution

The present thesis comprises several aspects of the automatic generation of program models for the formal verification of software. Moreover, it defines a new technique for verifying synchronization in real-world programs.

The main contributions of this thesis are the following.

- A simple formalization of CFG extraction from Java bytecode for the verification of temporal safety properties. The algorithm formally addresses all the complications of analyzing JBC, being the major two exceptions and virtual method resolution. It is based on BIR, a well-known intermediate bytecode representation. The extract algorithm comes with a soundness proof. Even though far from trivial, the proof is greatly simplified by a previous proof of an idealized extraction algorithm, being performed in terms of structural (i.e. finite state) rather than behavioural (i.e. infinite state) simulation w.r.t. the CFGs extracted by the latter.

- A formal framework for the extraction of control flow graphs from incomplete Java bytecode programs. We introduce a scheme for the modelling of incomplete programs, where unavailable components are represented by user-provided interfaces. Also, we generalize the previously-defined algorithm, and introduce rules to instantiate unavailable components that preserve the soundness of the extracted CFGs. It is proven that the algorithm extracts control flow graphs that structurally simulate any complete system that is implemented from the initial incomplete system. To the best of our knowledge, this is the first extraction algorithm which produces sound control flow graphs for the available components of an incomplete program. Also, it is the first one to produce control flow graphs which addresses the complications related to dynamic dispatching and exceptions for a real-world, object-oriented language in a modular set-up.
1.1. CONTRIBUTION

- **The Control Flow graph Extractor tool (ConFlEx)**, which implements the extraction algorithms for complete and incomplete Java bytecode programs. To the best of our knowledge, the tool is the first to implement a control flow extraction algorithm that is equipped with a correctness argument. Moreover, it is the first to soundly extract control flow graphs from incomplete programs. Thus, the tool is ideal for constructing models for formal software verification, especially for compositional verification of control flow safety properties.

- **SyncTask**, an intermediate imperative language to model bounded (though arbitrarily large) synchronization behaviours of programs with condition variables. We have defined a transformation of SyncTask programs into hierarchical Coloured Petri Nets, and reduce the proof of termination of SyncTask programs into a reachability analysis on the nets.

- An **annotation scheme** for assisting the extraction of program models from unbounded programs, which is generally undecidable. The scheme delimits a bounded subpart (synchronization) of the program behaviour, and thus makes the extraction (as a SyncTask program) decidable. Moreover, we reduce the verification of a liveness property that states the criterion for correct synchronization to the proof that the respective SyncTask program terminates. To the best of our knowledge, our framework is the first to prove a liveness property of multithreaded programs that synchronize via CVs.

- **The SyncTask Verifier (STaVe)**, which implements the framework for the verification of synchronization of multi-threaded programs. The tool parses annotated Java programs, and extracts their synchronization as a SyncTask program. It also translates a SyncTask program into a hierarchical Coloured Petri net, which are fed into CPN tools to establish the synchronization property. Parts of STaVe turned out to be useful for other projects, and became spin-offs. One was JavaParser2JCTree [22], a library that converts an abstract syntax tree in JavaParser representation into the OpenJDK’s Javac representation. Another part is the libcpntools [23] a library for generating coloured Petri nets in CPN Tool’s file format.

The work presented in this thesis resulted in the following peer-reviewed articles.


The following technical reports have been produced along the present work.

• A. Amighi, P. de C. Gomes, D. Gurov and M. Huisman. Provably Correct Control-Flow Graphs from Java Programs with Exceptions. KTH Royal Institute of Technology and University of Twente. 2012.


• P. de C. Gomes, D. Gurov and M. Huisman. Algorithmic Verification of Synchronization with Condition Variables. KTH Royal Institute of Technology. 2015.

The following Doctoral Licentiate thesis has presented some preliminary results also presented in the current thesis.

• P. de C. Gomes. Sound Modular Extraction of Control Flow Graphs from Java Bytecode. KTH Royal Institute of Technology. 2012.

The work present in this thesis has included the supervision or co-supervision of the following Master projects.


• J. Fogelström. Evaluation of a model checking back-end for STaVe: A study in performance between Petri Nets and Model Checking to correctness of parallel systems. KTH Royal Institute of Technology. 2016 (est.).

The following peer-reviewed articles have been published along the doctoral studies, but are not part of this thesis.

1.1. CONTRIBUTION


My contributions. The work presented in this thesis, and all the artifacts it has generated, are result of scientific collaborations. We now clarify the individual contributions, and the articles that have described each part.

The starting point was the idealized direct algorithm from Java Bytecode defined by Afshin Amighi, and its partial implementation in his extraction tool. I have revised the algorithm and notation. I proposed together with Afshin the extraction of CFGs using the BIR transformation, and we sketched the strategy used to prove the behavioural simulation. Next I defined formally the extraction algorithm, and presented the structural simulation proof, with the assistance of Marieke Huisman and Dilian Gurov. I was the main author of all publications about the extraction algorithm for complete systems [6, 4, 7, 8], but Dilian, Marieke and Afshin had equivalent contributions to mine. I re-wrote most of Afshin’s initial tool, and implemented the indirect algorithm for complete Java bytecode systems in the first version of ConFlEx [31].

I proposed and developed the generalization of the previous algorithm for incomplete systems. During the conception, I had several discussions with Attilio Picoco, who contributed with ideas, and insights about practical matters. Also, Dilian constantly revised and criticized my ideas. I discussed jointly with Attilio the design choices for extending ConFlEx to support incomplete systems. However, the majority of program code has been written by him, under my constant supervision. This is reflected in Attilio’s MSc. thesis [65], which focuses on the practical aspects of the tool implementation, while my Licentiate thesis focuses on the theoretical part [30]. I am the main author of the technical report [32] about the analysis of incomplete programs, but Attilio Picoco had an equivalent participation. I was the main author of the peer-reviewed article [33], with significant contributions from Dilian and Attilio.

The problem of verifying the correct synchronization of multithreaded programs with condition variables was suggested by Marieke Huisman. Under the constant criticism of both Marieke Huisman and Dilian Gurov, I have individually proposed and developed the verification framework. This includes the introduction of an intermediate representation and its design (SyncTask), an annotation scheme for Java programs, and the verification of the termination problems using CPNs. I have written the complete code for STaVe [24, 22, 23], and executed all the experimental evaluation. I am the main author of article [21] about the technique and tool, with significant participation of Dilian and Marieke.
1.2 Organization

The thesis is organized as follows. Chapter 2 summarizes results from previous works, which are necessary for the comprehension of our work. It also presents techniques that benefit from our work, and some motivating examples. Chapter 3 presents the extraction algorithm for complete Java bytecode systems, its correctness argument, and describes the implementation as ConFLex. Chapter 4 presents a formal framework to model incomplete Java bytecode programs, generalizes the extraction algorithm for this set-up, proves its correctness, and describes the extension of ConFLex to implement the algorithm. Chapter 5 presents a novel technique to verify the synchronization of multi-threaded Java programs using condition variables. It presents an annotation scheme to delimit the expected synchronization, its modelling as an intermediate language, and verification by means of Coloured Petri Nets. Finally, Chapter 6 summarizes our work and results.
Chapter 2

Background

This chapter summarizes existing definitions and results that have been used to establish the work presented in the thesis. The first one is the formal definition of Java virtual machine and bytecode by Freund and Mitchell [29]. We summarize it, and highlight the main aspects that influence the control flow. The second section presents the definitions of structure and behaviour of CFGs, as defined by Huisman et al. [39]. The third section presents an idealized extraction algorithm from Java bytecode, defined by Amighi [3]. We use this algorithm as a specification to prove the correctness of ours. The fourth work describes BIR, an intermediate representation of the Java bytecode, presented by Demange et al. [26]. Our algorithms use a transformation from Java bytecode into BIR because of its support for exceptions. The fifth section instantiates the notion of weak simulation introduced by Milner [61] for CFG structures. The definition is used to establish the correctness of extracted CFGs. The sixth section presents a compositional verification technique and tool-set introduced by Gurov et al. [35] that benefits from our control flow extraction, and illustrate the technique with some examples. We conclude by defining hierarchical Coloured Petri Nets, the model that we use to verify multithreaded programs.

2.1 Java Bytecode and the Java Virtual Machine

A compiler that targets Java bytecode generates class files, one for each declared class, or interface. Each class declaration contains a fully-qualified name, type information, and the declaration of methods and fields. We define Class-Name and Interface-Name to be the (countably) infinite sets of all class and interface names, respectively.

Bytecode programs use method references (Method-Ref), interface method references (Interface-Method-Ref) and field references (Field-Ref) to identify methods, interface methods and fields, respectively. These references are defined as triples of a static type (i.e., its most general type), a name (Label), and a type signature.
CHAPTER 2. BACKGROUND

(e.g., for a method, its return type and parameters). They are generated with the grammar in Figure 2.1

\[
\begin{align*}
\text{Method-Ref} & ::= \{\langle Class-Name, Label, Method-Type \rangle \}_{M} \\
\text{Interface-Method-Ref} & ::= \{\langle Interface-Name, Label, Method-Type \rangle \}_I \\
\text{Field-Ref} & ::= \{\langle Class-Name, Label, Field-Type \rangle \}_F
\end{align*}
\]

Figure 2.1: Grammar generating references

In this work we consider a subset of the instruction set described in [29]. Although the considered set is significantly smaller, it contains one representative from each group of instructions with similar behaviour w.r.t. the control flow. For example, we omit the `invokeinterface` instruction, since its control flow behaviour is analogous to the one for `invokevirtual`. The instructions `jsr q` and `ret r` for subroutine are not considered because they are deprecated since JBC version 1.6 [58]. Figure 2.2 shows the bytecode instruction set considered in our project. The symbol \( x \) denotes a local method variable, and \( p \) denotes an instruction address.

\[
\begin{align*}
\text{Instruction} & ::= \text{nop} | \text{push } c | \text{pop} | \text{dup} | \text{add} | \text{div} \\
& | \text{if } p | \text{goto } p \\
& | \text{load } x | \text{store } x \\
& | \text{new Class-Name} \\
& | \text{athrow} \\
& | \text{getfield Field-Type} | \text{putfield Field-Type} \\
& | \text{invokespecial Method-Ref} \\
& | \text{invokevirtual Method-Ref} \\
& | \text{vreturn} | \text{return}
\end{align*}
\]

Figure 2.2: Subset of the JBC instructions

Java bytecode is a stack-based executable language. That is, the operands for its instructions are stored on an operand stack, in contrast to a register-based approach. For example, the `if p` instruction branches to position \( p \) 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.

A JBC program is modeled as an environment \( \Gamma \), which is a partial map from class names, interface names and method signatures to their respective definitions. Figure 2.3 shows the definition of an environment \( \Gamma \). A class is defined by its parent class, the set of interfaces it implements, and its fields. An interface contains the set of interfaces it inherits from, and the set of methods it provides. Let \( \text{ADDR} \) be the set of all valid instruction addresses in \( \Gamma \). The body of a JBC method is a sequence of pairs of addresses and instructions. The sequence is non-empty, and the address
2.1. JAVA BYTECODE AND THE JAVA VIRTUAL MACHINE

Γ^I : Interface-Name → \{ interfaces : set of Interface-Name, method : set of Interface-Method-Ref \}

Γ^C : Class-Name → \{ super : Class-Name, interfaces : set of Interface-Name, fields : set of Field-Ref \}

Γ^M : Method-Ref → \{ code : (ADDR × Instruction)^+, handlers : Handler^* \}

Γ = Γ^I ∪ Γ^C ∪ Γ^M

Figure 2.3: Environment Γ of a Java program

of the first instruction is always zero. \( \text{Dom}(B) \subseteq \text{ADDR} \) is the set of valid program addresses for method \( m \), and \( B[k] \) denotes the instruction at position \( k \in \text{Dom}(B) \) in the method’s body. For convenience, \( m[k] = i \) denotes instruction \( i \) at location \( k \) of method \( m \).

Let \( \text{Excp-Name} \subseteq \text{Class-Name} \) be the (infinite) set of exceptions classes in Java. A method’s exception table is defined with quadruples of the form \( \langle b, e, t, x \rangle \), where \( b, e, t \in \text{ADDR} \) and \( x \in \text{Excp-Name} \). If an exception is thrown by an instruction with index \( i \) s.t. \( i \in [b, e) \) and it is from a subtype of \( x \), then \( m[t] \) is the first instruction of the corresponding handler. Thus, the instructions between \( b \) and \( e \) model the try block in a Java source program. The instructions starting at \( t \) model either the catch block that handles the exception \( x \), or a finally block, if \( x \) is from the special type any, defined as an alias of Throwable, the super-type of any exception.

The Java virtual machine contains a bytecode verifier (JBV), which performs several sanity checks on the code before the execution starts. For instance, it checks if methods terminate with either a return or an athrow instruction, and whether the branching instructions transfer the control to valid instruction addresses. The definition below states that a well-formed program is one that passes successfully all verification tasks. In the thesis we assume that the input bytecode is well-formed.

Definition 1 (Well-Formed Java Program). A well-formed Java bytecode program is a complete program which passes the JVM bytecode verification. The exhaustive list of verification tasks is presented in [76].

During the execution of the JVM, an active method, i.e., a method instance that has not terminated, is represented by an activation record. This is a quintuple that 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 an information \( z \) about the initialization of objects. The records are placed on the call stack, which stores in which sequence the methods are invoked. The top of the call stack contains the activation record of the method currently being executed, or the record \( (e)_{exc} \), representing the case when an exception is raised. Figure 2.4 shows the syntax for the call stack.

\[
A ::= A' \mid (x)_{exc} \cdot A' \\
A' ::= \langle m, p, f, s, z \rangle \cdot A' \mid \epsilon
\]

Figure 2.4: 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 invocation, 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 currently active methods. In summary: a JVM execution contains a single call stack, which, in turn, may contain several operand stacks.

An execution state of the Java virtual machine for a sequential program is defined as a configuration \( C = A; h \), where \( A \) is a call stack, and \( h \) represents a memory heap. The JVM behaviour is an 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 [29].

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 \( \Gamma^C\text{.interfaces} \) and \( \Gamma^C\text{.super} \), which contain the interfaces a class or an interface will extend, and in \( \Gamma^C\text{.super} \), 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, with the class \texttt{java.lang.Object} as root.

Inheritance is transitive in JBC programs. That is, one class or interface inherits in cascade from its immediate classes and interfaces. The subtyping relation is defined between classes or interfaces \( \tau_1 \) and \( \tau_2 \) in an environment \( \Gamma \). The relation holds whenever \( \tau_1 \) inherits transitively from \( \tau_2 \), and we denote this as \( \Gamma \vdash \tau_1 :<: \tau_2 \). Figure 2.5 shows the rules for the subtyping relation.

Subtyping plays a key role in the control flow analysis because of polymorphism, another OOP feature of bytecode. Polymorphism is the 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 to declare a method with the same signature, but with a different implementation. We call those virtual methods.

The invocation of virtual methods is executed by invokevirtual, which operates over two parameters. The first is a pair \( m \in \text{Method-Ref} \), which is hard-coded
2.1. JAVA BYTECODE AND THE JAVA VIRTUAL MACHINE

\[\begin{align*}
\text{[<:I REF]} & \quad \text{[<:I SUPER]} & \quad \text{[<:C REF]} \\
\omega \in \text{Interface-Name} & \quad \omega_2 \in \Gamma[\omega_2].\text{interfaces} & \quad \sigma \in \text{Class-Name} \\
\Gamma \vdash \omega <:_I \omega & \quad \Gamma \vdash \omega_1 <:_I \omega_2 & \quad \Gamma \vdash \sigma <:_C \sigma \\
\text{[<:C SUPER]} & \quad \text{[<:R CLASS]} & \quad \text{[<:R CLASS INT]} \\
\Gamma \vdash \sigma_1 <:_C \sigma_2 & \quad \Gamma \vdash \sigma_1 <:_R \sigma_2 & \quad \omega_1 \in \Gamma[\omega_2].\text{interfaces} \\
\Gamma[\sigma_2].\text{super} = \sigma_3 & \quad \Gamma \vdash \sigma_1 <:_C \sigma_2 & \quad \Gamma \vdash \omega_1 <:_I \omega_2 \\
\Gamma \vdash \sigma_1 <:_C \sigma_3 & \quad \Gamma \vdash \sigma_1 <:_R \sigma_2 & \\
\text{[<: INTERFACE]} & \quad \text{[<: REF]} \\
\Gamma \vdash \omega_1 <:_I \omega_2 & \quad \Gamma \vdash \tau_1 <:_R \tau_2 \\
\Gamma \vdash \omega_1 <:_\omega & \quad \Gamma \vdash \tau_1 <:_\tau \\
\end{align*}\]

Figure 2.5: Subtyping rules

in the bytecode. The second parameter is an object reference that is on top of the operand stack. The dynamic type of the object is what determines which of the polymorphic method implementations will be invoked. The exact dynamic type can only be determined at run-time. The only guarantee provided by the JBV is that the possible dynamic types are always subtypes of the static type. Different virtual method call (VMC) resolution algorithms can statically determine the set of the possible receivers for a given virtual invocation.

In JBC, exceptions are objects used to signal abnormal conditions during program execution. The exception classes are subtypes of java.lang.Throwable that are either defined in the standard Java API, or user-defined. An exception may 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 a reference to the exception to be thrown, which is on the top of the operand stack. Thus, static analyses have to perform some kind of stack evaluation to determine the possible types of the exception.

Upon an exception raising, the JVM searches for the first handler in the executing method’s exception table whose address range contains the address of the control point where the exception was raised, and its type is a subtype of the exception. If a suitable handler is found, the control is transferred to the first instruction in that block; otherwise the executing method is terminated abruptly, and the exception is propagated to the calling method, which now should handle the exception. This process continues until a method in the call stack handles the exception, or the program terminates.
Example 1 (Running Example of a Sequential Java Program). Figure 2.6 depicts a sample Java source program. It has a single class named `EvenOdd`, containing three methods. The methods’ control points are annotated in the left column. Figure 2.7 depicts the bytecode of the same program. The left column denotes the addresses of the methods’ instructions, which are also control points in the program execution. The present work covers only the analysis of Java bytecode. However, clearly the JBC representation is much more verbose than the source representation. Therefore, for understandability, we sometimes illustrate definitions using source code programs.

```java
public class EvenOdd {
    public static void main(String[] argv) {
        int myarg = Integer.parseInt(argv[1]);
        if (argv[0].equals("e"))
            even(myarg);
        else
            odd(myarg);
    }

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

    public static boolean even(int lx) {
        if (lx == 0)
            return true;
        else
            return odd(lx - 1);
    }
}
```

Figure 2.6: Example Java source program with control points

The entry method `main` receives two arguments upon invocation: the first one is a selector between methods `even` and `odd`; the second is the integer to be checked. It invokes two methods from the Java API: `parseInt` and `equals`. The method `odd` potentially throws an `ArithmeticException`. The method `even`, on the other hand, contains an exception handler for such an exception. If an `ArithmeticException` is raised in the interval of control points [0, 12) ([v15, v19) in the source), defined by the `try` block, then control is transferred to the control point 13 (v20 in source), which is the first instruction defined by the `catch` block.

2.2 Control Flow Graphs

Control flow graphs are program models where nodes represent the control points of a method, and the edges represent how instructions shift control between the points. In this work we are interested in a specific type of CFGs that abstract from all data, but preserve information about method invocations, and exceptions.
2.2. CONTROL FLOW GRAPHS

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

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

boolean even(int) {
    0: iload_0
    1: ifne 6
    4: iconst_1
    5: ireturn
    6: iload_0
    7: invokevirtual
        even(int)
    10: aload_1
    11: astore_1
    12: iconst_m1
    13: iload_0
    14: imul
    15: invokevirtual
        even(int)
    18: ireturn
}
```

Figure 2.7: Example program in Java bytecode

Other Java bytecode features are ignored. The following definitions are presented in [8]. These are slightly modified versions of the general notion of model, and the structure and behaviour of a CFG defined by Gurov et al. [35, 39].

**Definition 2** (Model, Initialized Model). A model is a state transition system \( M = (S, L, \rightarrow, A, \lambda) \) where \( S \) is a set of nodes, \( L \) a set of labels, \( \rightarrow \subseteq S \times L \times S \) a labelled transition relation, \( A \) a set of atomic propositions, and \( \lambda : S \rightarrow P(A) \) a valuation assigning the set of atomic propositions that hold at each node \( s \in S \). An initialized model is a pair \( (M, E) \), where \( E \subseteq S \) is a set of entry nodes.

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

The edges contain information about invocation instructions. We refer to edges corresponding to such instructions as visible, and label them with a method signature. Edges corresponding to other instructions are labelled with \( \epsilon \), and are called silent. Invocations of methods from the Java API are also considered silent, although their propagation of exceptions is taken into account.
Example 2 (CFG of a Java Program). Figure 2.8 shows the CFG extracted for the program in Example 1. We represent it by means of control points from the Java source, for simplicity. There is one sub-graph for each method in the program, and the nodes of each method are tagged with the method’s signature. Entry nodes are depicted by incoming edges without source.

There are several exceptional nodes in the CFG (named $e_1, e_2, \ldots$) that do not have a corresponding control point in the source code. They represent the configurations in which the control was taken by the JVM, to take care of an exception. Edges from an exceptional node to a normal one represent the presence of a handler for the exception at that control point. Exceptional nodes tagged with the atomic proposition $r$ denote the propagation of an exception by the method to a calling method.

The only visible edges are the ones relative to the invocations of methods even and odd. Notice that the invocation of `parseInt`, which is a method from the Java API, is considered to be a silent edge. However, the method’s signature declares that a `NumberFormatException` (N.F.E) is potentially propagated, and this is reflected by the edge to $e_1$.

Method graphs are the basic building blocks of control flow graphs. We define a method graph for sequential programs with procedures and exceptions as the instantiation of an initialized model, as follows.

**Definition 3** (Method Graph). A method graph with exceptions for a method $m \in \text{Method-Ref}$ over sets $M \subseteq \text{Method-Ref}$ and $E \subseteq \text{Excp-Name}$ is an initialized model $(M_m, E_m)$, where $M_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ with $V_m$ being the set of
2.2. CONTROL FLOW GRAPHS

control nodes of \( m, L_m = M \cup \{ \varepsilon \} \) the set of labels, \( A_m = \{ m, r \} \cup E \), \( m \in \lambda_m(v) \) for all \( v \in V_m \), and for all \( x, x' \in E \), if \( \{ x, x' \} \subseteq \lambda_m(v) \) then \( x = x' \), i.e., each control node is tagged with the signature of the method it belongs to and at most one exception. \( E_m \subseteq V_m \) is a non-empty set of entry control points of \( m \).

Control flow graphs come with an interface, which defines the methods that are provided and required. We say a CFG is closed if all the required methods are also provided; we say it is open otherwise. The interface also defines the exceptions that are potentially propagated by the provided methods. We should stress here that in the composition, having the definition of \( I^e \) as a pair assists us in tracking the method that propagates the exception. We define the notion of CFG interface as follows.

**Definition 4** (Control Flow Graph Interface). A Control Flow Graph interface is a triple \( I = (I^+, I^-, I^e) \) where \( I^+ \subseteq \text{Method-Ref} \) are the set of provided, and (externally) required methods, respectively. \( I^e \subseteq I^+ \times E \) is the finite set of potentially propagated exceptions by each provided method. The composition of interfaces is defined as \( I_1 \cup I_2 = (I_1^+ \cup I_2^+, (I_1^- \cup I_2^-), (I_1^e \cup I_2^e)) \).

A CFG is essentially a collection of method graphs. The composition of method graphs is defined as their disjoint union \( \biguplus \). We define a method’s CFG as the pair of its method graph and interface, and the control flow graph of a program is composed from the control flow graphs of all its methods. Now we formally define CFGs that model sequential programs with procedures and exceptions, as follows.

**Definition 5** (Control Flow Graph Structure). A Control Flow Graph \( G \) with interface \( I \), written \( G = (M, E) : I \) is inductively defined by:

- \( (M_m, E_m) : (\{ m \}, I^-, I^e) \) if \( (M_m, E_m) \) is a method graph for \( m \) over \( I^-, I^e \)
- \( G_1 \biguplus G_2 : I_1 \cup I_2 \) if \( G_1 : I_1 \) and \( G_2 : I_2 \)

**Example 3 (CFG interface and structure).** The method graph of \texttt{odd} is the central sub-graph in Figure 2.8 and its interface is \( (\{ \texttt{odd} \}, \{ \texttt{even} \}, \{ (\texttt{odd, Arithmetic-Exception}) \}) \). The composed CFG of the program is the disjoint union of all method graphs, as in Figure 2.8. Its interface is \( (\{ \texttt{main,odd,even} \}, \{ \}, \{ (\texttt{main, NumberFormatException}), (\texttt{main, ArithmeticException}), (\texttt{odd, ArithmeticException}) \}) \).

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

Definition 6 (CFG Behaviour). Let $\mathcal{G} = (\mathcal{M}, \mathcal{E}) : I$ be a closed control flow graph with exceptions such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The behaviour of $\mathcal{G}$ is described by the initialized model $b(\mathcal{G}) = (\mathcal{M}_b, \mathcal{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$ such that:

- $S_b \in V \times V^*$, i.e., states are pairs of control node and stack of control nodes,
- $L_b = \{\tau\} \cup L^C_b \cup L^X_b$ where $L^C_b = \{m_1 \mid m_2 \mid l \in \{\text{call}, \text{return}, \text{throw}\}, \lambda_1, \lambda_2 \in I^+\}$ (the set of call and return labels) and $L^X_b = \{l \mid l \in \{\text{throw}, \text{catch}\}, x \in \text{Excp-Name}\}$ (the set of exceptional transition labels).
- $A_b = A$
- $\lambda_b((v, \sigma)) = \lambda(v)$
- $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is the set of transitions induced by the following rules:

\[
\begin{align*}
\text{[transfer]} & \quad (v, \sigma) \xrightarrow{s_b} (v', \sigma) \quad \text{if } m \in I^+, v \xrightarrow{\lambda} v', r \notin \lambda(v), \\
\text{[call]} & \quad (v_1, \sigma) \xrightarrow{\text{call}} (v_2, v_1 \cdot \sigma) \quad \text{if } \{m_1, m_2\} \subseteq I^+, v_1 \xrightarrow{m_2} m_1 v'_1, \\
\text{[return]} & \quad (v_2, v_1 \cdot \sigma) \xrightarrow{\text{return}} (v'_1, \sigma) \quad \text{if } \{m_1, m_2\} \subseteq I^+, v_1 \xrightarrow{m_2} m_1 v'_1, \\
\text{[xreturn]} & \quad (v_2, v_1 \cdot \sigma) \xrightarrow{\text{xreturn}} (v'_1, \sigma) \quad \text{if } \{m_1, m_2\} \subseteq I^+, v_1 \xrightarrow{m_2} m_1 v'_1, \\
\text{[throw]} & \quad (v, \sigma) \xrightarrow{\text{throw}} (v', \sigma) \quad \text{if } m \in I^+, v \xrightarrow{\lambda} v', r \notin \lambda(v), x \in \text{Excp-Name}, x \notin \lambda(v), \\
\text{[catch]} & \quad (v, \sigma) \xrightarrow{\text{catch}} (v', \sigma) \quad \text{if } m \in I^+, v \xrightarrow{\lambda} v', r \notin \lambda(v), x \in \text{Excp-Name}, x \notin \lambda(v), \lambda(v') \cap \text{Excp-Name} = \emptyset.
\end{align*}
\]

The set of entry states is defined by $\mathcal{E}_b = \mathcal{E} \times \{\epsilon\}$, where $\epsilon$ is 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 control is transferred to the entry node of the callee method. A return-transition models the normal termination of a method: the calling node is popped from the stack, and control is transferred to the successor normal control node. An xreturn-transition models the abortion of a method execution by an uncaught exception x, and its propagation: the calling node is popped from the stack, and control is transferred to the successor exceptional node tagged with x.

Example 4 (CFG Behaviour). Consider the CFG in Figure 2.8. The following is an example run through the (infinite-state) behaviour induced by the CFG:

\[(v_1, \epsilon) \xrightarrow{\tau_b} (v_2, \epsilon) \xrightarrow{\tau_b} (v_3, \epsilon) \xrightarrow{\tau_b} (v_4, \epsilon) \xrightarrow{\text{main call even}} (v_{15}, v_4) \xrightarrow{\tau_b} (v_{16}, v_4) \xrightarrow{\tau_b} \]
\[(v_{18}, v_4) \xrightarrow{\text{even call odd}} (v_7, v_{18} \cdot v_4) \xrightarrow{\tau_b} (v_8, v_{18} \cdot v_4) \xrightarrow{\tau_b} (v_9, v_{18} \cdot v_4) \xrightarrow{\text{throw A.E.}} \]
\[(e_3, v_{18} \cdot v_4) \xrightarrow{\text{odd xreturn even}} (e_4, v_4) \xrightarrow{\text{catch A.E.}} (v_{20}, v_4) \xrightarrow{\tau_b} \ldots\]

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

2.3 Direct Extraction of CFGs from Bytecode

This section briefly describes the idealized direct extraction algorithm presented in [8]. The CFGs extracted with the algorithm have been proven to induce behaviours that simulate the JVM behaviour of the original program. We use this correctness result to establish the correctness of the extraction algorithms for both complete and incomplete JBC programs. We prove that the two algorithms structurally simulate the idealized one. Then, we conclude the correctness of our algorithms by reusing a previous result, which states that structural simulation implies behavioural simulation [35].

The idealized algorithm introduces two simplifications w.r.t. the set of JBC instructions. The first is the assumption of an oracle to provide information about exceptions. It lists which exceptions are potentially raised by the execution of each instruction type. Also, the instruction athrow (explicit exception throw) does not have an argument; instead the exception is determined at run-time by the top of the stack. Our algorithm replaces athrow with throw X, and uses the oracle to list the set X of possible exception types. Second, the instructions jsr q and ret r, for the invocation and return of subroutines, are not considered because they are deprecated since JBC version 1.6 [58].

The algorithm partitions the JBC instruction set into subsets of instructions with common behaviour w.r.t. the control flow. The classification enables a manageable list of extraction rules. Figure 2.9 presents the partitioned sets, an intuitive
### CHAPTER 2. BACKGROUND

<table>
<thead>
<tr>
<th>Subset</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPINST</td>
<td>Computational instructions</td>
<td><code>nop, push v, pop</code></td>
</tr>
<tr>
<td>CNDINST</td>
<td>Conditional instructions</td>
<td><code>ifeq q</code></td>
</tr>
<tr>
<td>JMPINST</td>
<td>Jump instructions</td>
<td><code>goto q</code></td>
</tr>
<tr>
<td>THRINST</td>
<td>Explicit exception throw</td>
<td><code>throw X</code></td>
</tr>
<tr>
<td>XMPINST</td>
<td>Potentially can raise exceptions</td>
<td><code>div, getfield f, new</code></td>
</tr>
<tr>
<td>INVINST</td>
<td>Method invocations</td>
<td><code>invokevirtual (o,m)</code></td>
</tr>
<tr>
<td>RETINST</td>
<td>Normal return instructions</td>
<td><code>return</code></td>
</tr>
</tbody>
</table>

Figure 2.9: Grouping of JBC instructions with common control flow behaviour

The description of the semantics, and examples of instructions that have a common control flow behaviour.

We now present how the idealized algorithm extracts a CFG, as defined in Section 2.2, from a given JBC program. First, it defines the elements of a CFG from a program’s method. The nodes of a method’s CFG are tagged with an address and a method signature. Based on Definition 3, to construct the nodes one has to specify $V_m$, $A_m$, $\lambda_m$, and $E_m$. A node $v \in V_m$ is uniquely identified by its control point $p \in \text{ADDR}$ and its atomic propositions, as tagged by the function $\lambda_m(v)$. The method signature is the default tag for all the method’s control nodes. If $m[p] \in \text{RETINST}$ then the node is additionally tagged with $r$. If the node is an exceptional node then it is tagged with the exception type $x \in E$. If $p = 0$ then the node will be a member of $E_m$.

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

**Definition 7** (CFG Extraction from JBC). The instruction-wise extraction function $G_{inc}: (\text{Method-Ref} \times \text{ADDR} \times \text{JBCInst}) \rightarrow P(V_m \times L_m \times V_m)$ is defined by the rules in Figure 2.10. The method graph for $m$ is defined as $G_{inc}(m) = \bigcup_{(p,i) \in B} G_{inc}^{m,p,i}$. The control flow graph for the program is defined as $G_{inc}(\Gamma) = \bigcup_{(m|\Gamma[m] \in \Gamma)} G_{inc}(m)$.

We start the explanation of the construction rules with the auxiliary functions. They compute important information for the control flow analysis, such as exception raising, exception handling or propagation, and virtual method calls.

**Auxiliary functions** The function $\text{succ}(p)$ takes a control point $p$, and yields the successor control point. We define $\text{jmp}(i)$, which receives an instruction of types
2.3. DIRECT EXTRACTION OF CFGS FROM BYTECODE

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

\[
E_m^p = \bigcup_{x \in X(m[p])} \mathcal{H}_{x,p}^m
\]

\[
N_{p,n}^m = \bigcup_{\{x \in X(n[p])\}} \mathcal{H}_{x,p,n}^m
\]

\[
Rec_i^p = \begin{cases} 
\{\text{static}(\circ_i)\} & \text{if } i \in \{\text{invokespecial } (o,n), \text{ invokestatic } (o,n)\} \\
\{n_x \mid \tau \in \text{restr}(o,n)\} & \text{if } i \in \{\text{invokevirtual } (o,n), \text{ invokeinterface } (o,n)\}
\end{cases}
\]

\[
G_{nc}^{m,i} = \begin{cases} 
\{(\circ_m^p, \varepsilon, \circ_m^{nc(p)})\} & \text{if } i \in \text{CmpInst} \\
\{(\circ_m^p, \varepsilon, \circ_m^{jmp(i)})\} & \text{if } i \in \text{JmpInst} \\
\{(\circ_m^p, \varepsilon, \circ_m^{nc(p)}), (\circ_m^p, \varepsilon, \circ_m^{jmp(i)})\} & \text{if } i \in \text{CndInst} \\
\{(\circ_m^p, \varepsilon, \circ_m^{nc(p)})\} \cup E_m^p & \text{if } i \in \text{XmpInst} \\
\bigcup_{p \in X} \mathcal{H}_{x,p,n}^m & \text{if } i \in \text{ThInst} \\
\bigcup_{p \in Rec_i^p} \{(\circ_m^p, n_x, \circ_m^{nc(p)})\} \cup \mathcal{H}_{x,p}^m \cup N_{p,n}^m & \text{if } i \in \text{InvInst} \\
\emptyset & \text{if } i \in \text{RetInst}
\end{cases}
\]

Figure 2.10: CFG construction rules
The auxiliary function $E$ uses $H$ to compute exceptional edges for all exceptions that can potentially be raised by a given instruction. The function $X : XmpInst \to \mathcal{P}(Excp-Name)$ returns the set of run-time exceptions that an instruction may raise. The throw instruction is handled similarly, where $X$ is the set of possible exceptions. Both rely on an oracle to list the set of exceptions. Function $N^p,n$ generates the set of edges to handle all uncaught exceptions from the possible callee $n$. Notice that it performs an inter-procedural analysis, since it evaluates the exceptional return nodes from the callee method’s CFG.

To extract edges for method invocations, the auxiliary function $Rec_i^\Gamma$ yields the set of possible method signatures of a method call in environment $\Gamma$. The receiver object for invokevirtual is determined by late binding. For this, the virtual method call resolution function $res_\alpha^\Gamma$ is employed, where $\alpha$ is a parameter denoting an external standard static analysis to resolve the call. We use $\eta_T$ to indicate method $n$ from class $T$. For example, Rapid Type Analysis (RTA) [10] returns the set of subtypes of the caller’s static type which are instantiated in the program (created by a new instruction). So, for $\alpha = RTA$, the result of the resolution for object $o$ and method $n$ in environment $\Gamma$ will be:

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

where $IC_\Gamma$ is the set of instantiated classes in environment $\Gamma$, $staticT(o)$ gives the static type of object $o$, and $lookup(n,\tau)$ corresponds to the signature of $n$ in $\tau$, i.e., $\tau$ is a subtype of $o$’s static type and method $n$ is defined in class $\tau$.

**Construction rules** For simple computational instructions, a direct edge to the next control address is established. For jump instructions, an edge to the jump address ($jmp(i)$, for instruction $i$) is generated. For conditional instructions, edges to the next control address and to the address specified for the jump are generated. For instructions in XmpInst, edges for all possible flows are added: successful execution and exceptional execution, with edges for successful and failed exception handling, as defined by function $H^{x,p,l}_m$. Required edges for instruction throw can simply be produced by using function $H$ for all the exceptions in the over-approximated instruction parameter $X$.

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

In all the rules, if the target node points to an instruction $i \in RetInst$ then the node will be tagged with $r$. To keep the presentation of the rules reasonably
2.4. THE BIR LANGUAGE

simple we do not show the target node checks in Figure 2.10. Thus the rule for $i \in \text{RetInst}$ does not generate any edges in the CFG.

We now enunciate the soundness theorem for the idealized algorithm, which is later referenced in Section 3.2. We refer to [8] for the complete proof.

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

### 2.4 The BIR Language

The BIR language is an intermediate representation of Java bytecode [26]. The main difference with JBC is that BIR instructions are stack-less, in contrast to bytecode instructions that operate over values stored on the operand stack.

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

Figure 2.11 summarizes the BIR syntax. Its instructions operate over expression trees, i.e., arithmetic expressions composed of constants, operations, variables, and fields of other expressions ($expr_1$). BIR does not have operations over strings and booleans; these are transformed into method calls by the BC2BIR transformation. The transformation also reconstructs expression trees, i.e., it collapses one-to-many stack-based operations into a single expression. As a result, a JBC program represented in BIR typically has fewer instructions than the original program.

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

Many of the BIR instructions have an equivalent JBC counterpart, e.g., `nop`, `goto` and `if`. A `return expr` ends the execution of a method with a return value, while `return` ends a `void` method. Method call instructions are represented by their method signature. For non-`void` methods, the instruction assigns the result value to a variable. In contrast to JBC, object allocation and initialization happen in a single step, during the execution of BIR’s `new` instruction, which also performs a call to an object constructor.

The `throw` instruction explicitly transfers control flow to the exception handling mechanism, similarly to the `athrow` instruction in JBC. Notice that the BIR instruction also has an operand, similar to the `throw X` instruction introduced in Section 2.3. Here, a sound static analysis that over-approximates the possible types of the operand (e.g., see [10]) implements the oracle for explicit exceptions.
op \text{er} ::= c \mid \text{null} \quad \text{(constants)}
| f \quad \text{(field name)}
| C \quad \text{(class name)}
| \text{pc} \mid \text{pc'} \quad \text{(program counter)}
| m \mid n \mid n' \quad \text{(method name)}

l\text{var} ::= \ell_0 \mid \ell_1 \mid \ldots \ell_j \quad \text{(local vars.)}

t\text{var} ::= \ell_0 \mid \ell_1 \mid \ldots \ell_k \quad \text{(temp. vars.)}

t\text{arget} ::= l\text{var} \mid t\text{var} \mid \text{expr.f} \mid \text{this}

\text{expr} ::= c \mid \text{null} \mid \text{target} \mid \text{expr} \oplus \text{expr}

\text{Assignment} ::= \text{target} := \text{expr}

\text{Return} ::= \text{return expr} \mid \text{return}

\text{MethodCall} ::= \text{expr.n(expr,...,expr)} \mid \text{target} := \text{expr.n(expr,...,expr)}

\text{NewObject} ::= \text{target} := \text{new C(expr,...,expr)}

\text{Assertion} ::= \text{nonnull expr} \mid \text{notzero expr} \mid \ldots

\text{BirInstr} ::= \text{nop} \mid \text{if expr pc} \mid \text{goto pc} \mid \text{throw expr} \mid \text{mayinit C}
| \text{Assignment} \mid \text{Return} \mid \text{MethodCall} \mid \text{NewObject} \mid \text{Assertion}

Figure 2.11: Expressions and instructions of BIR

BIR’s support of implicit exceptions follows the approach proposed for the Jalapeño compiler [13]. It inserts special assertions before the instructions that can potentially raise an exception, as defined by the JVM. Thus, the BIR transformation implements the oracle described in Section 2.3 for the exceptions raised implicitly by the instructions’ execution. Java bytecode also has class initializers, i.e., the one-time initialization procedures of a class’s static fields, invoked when the first object of a class is allocated by a JBC instruction new. If some exception is raised inside the initializer, the JVM captures it, and raises an Exception-InInitializerError. Thus, BIR adds a special instruction, called mayinit, to indicate that at that point a class initializer may be invoked.

Figure 2.12 shows all implicit exceptions that are currently supported by the BC2BIR transformation [11], and the associated assertion. For example, the transformation inserts a notnull assertion before any instruction that might raise a NullPointerException, such as an access to a reference. If the assertion holds, it behaves as a nop, and control flow passes to the next instruction. If the assertion fails, control flow is passed to the exception handling mechanism.

Next, we give a short overview of the BC2BIR transformation. It translates a complete JBC program into BIR by symbolically executing the bytecode using an abstract stack. This stack is used to reconstruct expression trees and to connect
2.4. THE BIR LANGUAGE

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>[notzero]</td>
<td>ArithmeticException</td>
</tr>
<tr>
<td>[checkbound]</td>
<td>ArrayIndexOutOfBoundsException</td>
</tr>
<tr>
<td>[checkstore]</td>
<td>ArrayStoreException</td>
</tr>
<tr>
<td>[checkcast]</td>
<td>ClassCastException</td>
</tr>
<tr>
<td>[mayinit]</td>
<td>ExceptionInInitializerError</td>
</tr>
<tr>
<td>[notneg]</td>
<td>NegativeArraySizeException</td>
</tr>
<tr>
<td>[nonnull]</td>
<td>NullPointerException</td>
</tr>
</tbody>
</table>

Figure 2.12: Implicit exceptions supported by BIR, and associated assertions

instructions to their operands. As we are only interested in the set of BIR instructions that can be produced, we do not discuss all details of this transformation. For the complete transformation algorithm we refer to [26].

The symbolic execution of the individual instructions is defined by a function $\text{BC2BIR}_{\text{instr}}$ that, given a program counter, a JBC instruction and an abstract stack, outputs a set of BIR instructions and a modified abstract stack. In case there is no match for a pair of bytecode instruction and stack, the function returns the $\text{Fail}$ element, and the algorithm aborts. The function $\text{BC2BIR}_{\text{instr}}$ is defined as follows.

**Definition 8 (From JBC to BIR).** Let $\text{AbsStack} \in \text{expr}^*$. The rules defining the instruction-wise transformation $\text{BC2BIR}_{\text{instr}} : \mathbb{N} \times \text{JbcInstr} \times \text{AbsStack} \rightarrow (\text{BirInstr}^* \times \text{AbsStack}) \cup \{\text{Fail}\}$ from JBC into BIR are given in Figure 2.13.

As a convention, we use square brackets to distinguish BIR instructions from their JBC counterpart. The variables $t_k$ are new and are introduced by the transformation.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>∅</td>
<td>nop</td>
<td>[nop]</td>
<td>div</td>
<td>[notzero e]</td>
</tr>
<tr>
<td>push c</td>
<td>∅</td>
<td>if p</td>
<td>[if e pc’]</td>
<td>athrow</td>
<td>[throw e]</td>
</tr>
<tr>
<td>dup</td>
<td>∅</td>
<td>goto p</td>
<td>[goto pc’]</td>
<td>new C</td>
<td>[mayinit C]</td>
</tr>
<tr>
<td>load $l_k$</td>
<td>∅</td>
<td>return</td>
<td>[return]</td>
<td>getfield f</td>
<td>[nonnull e]</td>
</tr>
<tr>
<td>add</td>
<td>∅</td>
<td>vreturn</td>
<td>[return e]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

store $l_j$ | $[l_j:=e]$ or $[t_k:=l_j;l_j:=e]$ |
putfield $f$ | $[\text{nonnull } e;FSave(pc,f,as);e.f:=e’]$ |
invokevirtual $n$ | $[\text{nonnull } e;HSave(pc,as);t_k:=e.n(...)]$ |
invokespecial $n$ | $[\text{null;HSave(pc,as);t_k:=new C(...)}]$ if n=C $[\text{null;HSave(pc,as);t_k:=e.n(...)}]$ otherwise

Figure 2.13: Rules for $\text{BC2BIR}_{\text{instr}}$
JBC instructions if, goto, return and vreturn are transformed into corresponding BIR instructions. The new instruction is different from [new C()] in BIR, it produces only a [mayinit] for static initialization of C. As mentioned above, object allocation in BIR happens at the same time as initialization, i.e., when the constructor is called. The getfield f instruction reads a field from the object reference at the top of the stack. This might raise a NullPointerException, therefore the transformation inserts a [nonnull] assertion.

The store x instruction produces one or two assignments, depending on the state of the abstract stack. Instruction putfield f outputs a set of BIR instructions: [nonnull e] guards whether e is a valid reference; then the auxiliary function FSave introduces a set of assignment instructions to temporary variables; followed by the assignment to the field (e.f). Similarly, instruction invokevirtual generates a [nonnull] assertion, followed by a set of assignments to temporary variables – represented as the auxiliary function HSave – and the call instruction itself. The transformation of invokespecial can produce two different sequences of BIR instructions. In the first case, there are assignments to temporary variables (HSave), followed by the instruction [new C] which denotes a call to an object constructor C. The second case is the same as for invokevirtual.

Figure 2.14 shows the JBC and BIR versions of method boolean odd(int) from Figure 2.6. The different colors show the collapsing of instructions by the transformation; the underlined instructions are the ones that produce BIR instructions. The BIR method has a local variable (l0) and two newly introduced variables (t0 and t1). Notice that the argument for the method invocation and the operand to the [if] instructions are reconstructed expression trees. The [nonnull] instruction asserts that a NullPointerException can potentially be raised in the program point 3. The [mayinit] instruction shows that class ArithmeticException can be initialized at that program point.

2.5 Compositional Verification of Control Flow Properties

In this section we illustrate the utility of the extracted CFGs by briefly describing the application and context that motivated the algorithms and tools presented in Chapters 3 and 4. However, we should stress that the extracted CFGs are also useful for other types of program analyses, such as [34, 9, 78].

Gurov et al. developed a technique for the verification of control flow based temporal safety properties, using CFGs as a program model [35]. The properties are specified in Simulation Logic, which is the (safety) fragment of the modal μ-calculus [53] without diamond modalities and least fixed points. The correctness of the verification results is therefore only guaranteed for models that are sound w.r.t. this class of properties. The framework has been extended to also support the more intuitive (linear-time) temporal logic Weak LTL, which is the (safety) fragment of LTL [66] that uses the weak version of the until temporal operator.

With the technique of [35] one can thus verify properties over sequences of
2.5. COMPOSITIONAL VERIFICATION

public static boolean odd(int n)

Java bytecode

<table>
<thead>
<tr>
<th></th>
<th>BIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>iload_0</td>
</tr>
<tr>
<td>1</td>
<td>ifge 12</td>
</tr>
<tr>
<td>4</td>
<td>new ArithmeticException</td>
</tr>
<tr>
<td>7</td>
<td>dup</td>
</tr>
<tr>
<td>8</td>
<td>invokevirtual ArithmeticException()</td>
</tr>
<tr>
<td>11</td>
<td>athrow</td>
</tr>
<tr>
<td>12</td>
<td>iload_0</td>
</tr>
<tr>
<td>13</td>
<td>ifne 18</td>
</tr>
<tr>
<td>16</td>
<td>iconst_0</td>
</tr>
<tr>
<td>17</td>
<td>ireturn</td>
</tr>
<tr>
<td>18</td>
<td>iload_0</td>
</tr>
<tr>
<td>19</td>
<td>iconst_1</td>
</tr>
<tr>
<td>20</td>
<td>isub</td>
</tr>
<tr>
<td>21</td>
<td>invokevirtual even(int)</td>
</tr>
<tr>
<td>24</td>
<td>ireturn</td>
</tr>
</tbody>
</table>

Figure 2.14: Comparison between instructions in method boolean odd(int)

method invocations and exceptions, using the CFGs extracted with the algorithms presented here. Examples of useful properties are: “Exception X will only be caught by a certain method M”, “If exception X is thrown, the first invoked method must be the state-restoring method M”, and “Exception X is always caught within the method that raised it, or by its caller method.”

The technique is tailored for compositional verification, but can also be used in a non-compositional setting. It decouples the global correctness of a system from its implementation; the correctness is relativized over its components’ local specifications. A global property is checked over a maximal control flow graph, constructed from the software components’ properties, and interfaces. A maximal control flow graph for a given property is the CFG that simulates any CFG that satisfies the property. Then, the verification of the local properties over the implementation of each component guarantees the global correctness.

Now we present the compositional verification technique. Let $\psi_i$ be the specification of a software component $i$, and $I_i$ its interface. A maximal control flow graph for the pair is denoted as $Max(\psi_i, I_i)$. Also, let $G_i$ be the control flow graph from a given component, $k$ be the number of components for an arbitrary system, and $\phi$ be a global property. The compositional verification principle is presented as the following proof rule, with $k + 1$ premises.
CHAPTER 2. BACKGROUND

The CFG of a complete program, as defined in Section 2.2, is the disjoint union of the CFGs from its components. Thus, the principle states that the program’s control flow graph satisfies the global specification if the latter is satisfied by the system’s composed maximal control flow graph, and if each component’s CFG satisfies its local specification.

The verification technique is implemented as CVPP [40], a tool set for the (compositional) verification of JBC programs w.r.t. temporal safety properties of sequences of method invocations, and exceptions. CVPP is wrapped by ProMoVer [74, 73], a tool that automatically encapsulates the verification steps.

In CVPP/ProMoVer, the compositional verification principle gives rise to two parallel tasks: (i) the verification of local properties over CFGs from methods, and (ii) the verification of the global property over the maximal control flow graph. The local properties of software components are structural, i.e., reason about the CFG structure (Def. 5). The global property is behavioural, i.e., reasons about the CFG behaviour (Def. 6). The two independent tasks are executed as follows.

(i) Check $G_i ⊨ \psi_i$ for $i = 1, \ldots, k$: (a) extract CFGs $G_i$ for each component, and (b) model check $G_i$ against $\psi_i$. CFGs are represented as finite automata, and verified by means of standard finite-state model checking.

(ii) Check $\bigcup_{i=1}^{k} \text{Max}(\psi_i, I_i) \models \phi$: (a) for all method specifications $\psi_i$ and interfaces $I_i$, construct maximal flow graphs $\text{Max}(\psi_i, I_i)$, (b) compose the graphs, resulting in the maximal control flow graph $G_{\text{Max}}$, and (c) model check $G_{\text{Max}}$ against the global property $\phi$. For (c), the behaviour of $G_{\text{Max}}$ is represented as a push-down system (PDS), and a PDS model checker is used.

The verification of a complete Java bytecode program may be performed non-compositionally over the extracted CFGs, instead of computing the maximal flow graphs. The first step is the extraction of the program’s CFG. Next, the CFG is used to construct a push-down system (PDS) that represents the induced CFG behaviour, following the operational semantics from Definition 6. Finally, CVPP verifies the temporal property of interest against the PDS by using a standard PDS model checker [50, 69].

Example 5 (Verification of Complete Programs). Suppose we wish to verify whether the program in Figure 2.7 will never abort because of an uncaught Arithmetic-Exception. The following formula in Weak LTL expresses this property, where $G$ is the temporal operator globally (or always). The formula essentially states that program control never reaches a return point of method main tagged with the given exception type:

$$\phi_1 = G \neg (\text{main} \land r \land \text{ArithmeticException})$$
CVPP extracts the CFG from the program, and creates a PDS that represents its behaviour. Both the PDS and \( \phi_1 \) are inputs to the PDS model checker. The checker establishes that \( \phi_1 \) does not hold for the CFG behaviour. A counter-example produced by the checker exhibits a run that violates \( \phi_1 \): initially main invokes odd, then odd throws, but does not catch, an ArithmeticException, the exception is then propagated to main, and finally causes its exceptional termination.

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

\[
\phi_2 = (\text{main W even}) \rightarrow G\neg(\text{main} \land r \land \text{ArithmeticException})
\]

Again, CVPP feeds \( \phi_2 \) and the PDS into the model checker, which shows that \( \phi_2 \) holds for the behaviour induced by the CFG from the program in Example 2.

CVPP/ProMoVer also support the verification of incomplete systems. For example, assume that one wants to verify an incomplete Java bytecode system with \( k \) components. However, the component \( i \) (\( 1 \leq i \leq k \)) is missing. Still the global correctness of the system can be verified by the task (ii). The local correctness of the available components can be verified immediately. The only pending verification task would be the local verification of \( G_i = \psi_i \), which is postponed until the arrival of component \( i \).

The mentioned scenario applies for systems that depend on a component provided by a third-party. For instance, an ATM system where a component is provided in the smart card inserted by the user. One can verify in advance a system’s global property, and the local properties of the available components. The pending task is the verification of the component in the smart-card, which is light-weight, and can be delayed until the user arrives to the ATM. If the verification fails, then the ATM denies any transaction.

**Example 6 (Verification of Incomplete Programs).** Let’s suppose that the program in Example 1 is incomplete, and let’s remove the code of method even in Figure 2.9. Still, one may establish global properties over the composed model, and local properties over the CFGs extracted from the available methods. The pending task is the verification of the unavailable method even, which is performed once its code is provided.

We specify even with the interface \( I_{\text{even}} = (\{\text{even}\}, \{\text{odd}\}, \{\}) \). It declares that the method may call itself or odd, and does not propagate any exceptions. We define informally the local property \( \psi_{\text{even}} \) for the missing method as “after calling odd, even must terminate normally”, and construct the maximal CFG for \( \psi_{\text{even}} \) and \( I_{\text{even}} \). Also, we extract the CFGs from the available methods main and odd. In this specific example, the method graphs for available methods are the same as in Figure 2.8. Then, we compose them with the maximal CFG for even.
We verify the global property $\phi_2$ once again, now in the set-up with unavailable code. It 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 $\psi_{\text{even}}$. If it holds, the correctness of the program is established w.r.t. $\phi_2$.

Let us now consider another global property $\phi_3$, defined informally as “A method invoked by even cannot raise a ClassCastException”. The property is checked over the same composed model. However, $\phi_3$ does not hold since neither the interface, nor the local property restrict a ClassCastException from being raised by even; the restriction is for the propagation of the exception. Thus, even may call itself, raise the exception, but catch it locally. 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.

2.6 Weak Simulation on Models

The results presented here follow closely Milner [61], adapted to initialized models and thus to CFGs. As usual, in the context of a transition relation $\rightarrow$, we shall write $p_i \xrightarrow{\beta} p_j$ to denote $(p_i, l, p_j) \in \rightarrow$. Also, we use $\varepsilon$ to label silent edges (instead of $\tau$ used in Milner [61], to be consistent with our notation for models). The transition relation $\rightarrow$ induces a weak labelled transition relation in the standard fashion, where $\beta \neq \varepsilon$:

\[
\begin{align*}
\Rightarrow & \overset{\text{def}}{=} \varepsilon^* \\
\beta \Rightarrow & \overset{\text{def}}{=} \Rightarrow \beta \Rightarrow
\end{align*}
\]

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

**Definition 9** (Weak Simulation on Models). Let $(M_{p}, E_{p})$ and $(M_{q}, E_{q})$ be two initialized models, with $M_{p} = (S_{p}, L, \rightarrow_{p}, A, \lambda_{p})$ and $M_{q} = (S_{q}, L, \rightarrow_{q}, A, \lambda_{q})$, and let $R \subseteq V_{p} \times V_{q}$. Then $R$ is a weak simulation if for all $(p_i, q_i) \in R$ the following conditions hold:

1. $\lambda_{p}(p_i) = \lambda_{q}(q_i)$;
2. if $p_i \Rightarrow p_j$ then there is $q_j \in V_{q}$ such that $q_i \Rightarrow q_j$ and $(p_j, q_j) \in R$;
3. if $p_i \Rightarrow^\beta p_j$ then there is $q_j \in V_{q}$ such that $q_i \Rightarrow^\beta q_j$ and $(p_j, q_j) \in R$.

We say that $q$ weakly simulates $p$ if $(p, q) \in R$ for some weak simulation relation $R$. We also say that $(M_{q}, E_{q})$ weakly simulates $(M_{p}, E_{p})$ if for every $p \in E_{p}$ there is $q \in E_{q}$ such that $q$ weakly simulates $p$.

The following proposition (again in the style of [61]) allows for more compact simulation proofs.
Proposition 1. Let \((M_p, E_p)\) and \((M_q, E_q)\) be two initialized models, with \(M_p = (S_p, L, \rightarrow_p, A, \lambda_p)\) and \(M_q = (S_q, L, \rightarrow_q, A, \lambda_q)\), and let \(R \subseteq V_p \times V_q\). Then \(R\) is a weak simulation if for all \((p_i, q_i) \in R\) the following conditions hold:

1. \(\lambda_p(p_i) = \lambda_q(q_i)\);
2. If \(p_i \xrightarrow{p} p_j\) then there is \(q_j \in V_q\) such that \(q_i \xrightarrow{q} q_j\) and \((p_j, q_j) \in R\);
3. If \(p_i \xrightarrow{q} p_j\) then there is \(q_j \in V_q\) such that \(q_i \xrightarrow{q} q_j\) and \((p_j, q_j) \in R\).

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

2.7 Hierarchical Coloured Petri Nets

We now introduce Coloured Petri Nets, the model used in the verification technique presented in Chapter 5 for checking the correct synchronization with condition variables. The technique and theoretical results are presented informally. Thus, here we briefly describe and illustrate the CPN concepts, and refer to [45, Chapters 4,6] for the formal definitions.

Petri Nets (PN) are bipartite directed graphs where nodes are either places, visually represented as ellipses, or transitions, represented as rectangles. Arcs connect places to transitions, and vice versa. A place contains a non-negative number of tokens, which we refer to as its marking. A PN configuration consists of a distribution of tokens over the places, and it is also commonly referred to as a net marking. We remove the ambiguity along the thesis, and use the term ‘marking’ only when referring to places.

A transition is enabled if all its incoming places are marked, i.e., there is at least one token in all its input places. An enabled transition may fire, i.e., atomically consume tokens from the input places and produce tokens on the output places. The choice of which of the enabled transitions fires is non-deterministic.

Inhibitor arcs extend PNs by enabling a transition if their incoming places are empty. Or equivalently, they ‘inhibit’ a transition if they are marked by a token. These arcs are useful for testing emptiness of a place, meaning, for example, the exhaustion of some resource. Inhibitor arcs are depicted with a bubble, instead of an arrow. PNs with inhibitor arcs are more expressive, and reachability becomes generally undecidable. However, if a PN with inhibitor arcs is bounded, meaning that there exists a finite upper-bound for the tokens in the net, then reachability is still decidable [14]. In this thesis we extract bounded nets; thus we still stay in the decidable fragment.

Hierarchical Coloured Petri Nets [42] extend plain PNs with data. They declare colour sets, and assign one for each place. Transitions are enabled if all input places contain at least one token of the same colour as the incoming arc. CPNs generalize standard PNs. That is, a PN is simply a CPN with a single colour.
Figure 2.15: A hierarchical CPN representing the flow of passengers

CPNs provide a modular concept called subpage for the declaration and instantiation of components, which is analogous to subroutines in procedural languages. A subpage receives and returns tokens on its in- and out-port places, similarly to a procedure receiving parameters and returning a value. We depict ports as doubly outlined ellipses with the direction indicated on the lower corner. The instantiation of a subpage is modelled as a substitution transition (ST), and is analogous to a procedure invocation. It has incoming and outgoing arcs from and to its in- and out-socket places, which are assigned to matching in- and out-ports, respectively, in the subpage. STs are depicted as doubly outlined rectangles, with instantiated subpage name on the bottom.

Finally, fusion places are another modular concept, which are analogous to global variables. These enable the instantiation of the same place in several STs, and are graphically represented with a centralized rectangle with the place’s name.

Example 7 (Hierarchical Coloured Petri Net). Figure 2.15 shows a hierarchical CPN net that models a flow of passengers on an airplane. Passengers are separated into two categories: VIP, which have higher priority to board, and NORMAL, which have low priority. Upon boarding, the airline increments the number of passengers, so it can provide the exact amount of meals. After boarding, there is no distinction of
treatment between the passengers w.r.t. being served a meal, or exiting the plane. Despite being an over-simplified model, the CPN contains all concepts mentioned above, and we now explain them.

The CPN contains two pages. The first is the Queue top page, which defines the queuing and boarding of passengers. The services provided to the passengers after they have boarded are modelled with the substitution transition Service, which has Boarded and Landed as in- and out-sockets, respectively. The subpage Service models the passengers’ service and landing. For simplicity, the subpage has been named to the ST that instantiates it, just like its in- and out-ports have been named to the in- and out-sockets that they are assigned to.

The colour set CLIENT defines the two passenger types. VIP passengers queue in the High Priority place and are initially fifteen, as denoted by its marking on the top; NORMAL passengers are queued in the Low Priority place, with marking containing fifty five NORMAL tokens. The Board High transition is enabled as long as there are tokens in High Priority, while Board Low is disabled by the inhibitor arc. Moreover, whenever either of the transitions is enabled, it adds a token of colour FOOD to the fusion place Meals. Notice that the place is present in both Queue and Service, and represents the same entity. That is, the addition or removal of a token from Meals in one of the pages is reflected in the other.
Chapter 3

CFG Extraction from Complete Programs

This chapter presents the two-phase transformation from (complete) Java bytecode programs into control flow graphs, using the \texttt{BC2BIR} transformation, presented in Section 2.4. First we define the extraction function of CFGs from BIR, which we call $\mathcal{G}_{\text{bir}}$. Next, we outline the correctness proof of this indirect algorithm: CFGs extracted with $\texttt{BC2BIR} \circ \mathcal{G}_{\text{bir}}$ structurally simulate CFGs extracted with the $\mathcal{G}_{\text{jbc}}$ algorithm, presented in Section 2.3. We illustrate the proof by presenting one interesting case, and refer to the complete proof in Appendix A. We conclude by presenting the proof strategy, which shows that CFGs extracted indirectly from a JBC program simulate the program behaviour.

The work presented in this chapter has been published in [7, 8].

3.1 Extraction of CFGs from BIR

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

To define $\mathcal{G}_{\text{bir}}$, we introduce some auxiliary functions, which are similar to the ones introduced for the direct extraction (in Section 2.3). As a convention, we use bars (e.g., $\overline{N}$) to differentiate the corresponding functions from the direct, and indirect algorithms.

The auxiliary function $\overline{k}_{\text{pc},x,m}$ returns the first handler (if any) for the exception of type (or a subtype of) $x$ at position $\text{pc}$. The function $\overline{\chi}_i$ returns the exceptions associated with an instruction $i$, as presented in Figure 2.12. The function $\overline{H}_{\text{pc},x,l}$ queries the function $\overline{k}$ for exception handlers in the control point $\text{pc}$ for the exception type $x$: if there is any, it returns two edges: one from a normal to an exceptional control node, and one from the exceptional node to the normal node tagged with
the handler’s initial control point; otherwise, it returns an edge to an exceptional return node. The function also receives a label \( l \) as argument, which may be \( \varepsilon \), or a method signature.

The extraction is parametrized on a virtual method call resolution algorithm \( \alpha \), in the same fashion as the \( \mathcal{G}_{\text{inc}} \) algorithm presented in Section 2.3. The function \( \text{res}^{\alpha}(n') \) uses \( \alpha \) to return a safe over-approximation of the possible receivers to a virtual invocation of a method with signature \( n' \), or the single receiver if the signature is from a non-virtual method (e.g., a static method).

Let \( \Gamma_b \) be the environment modeling the BIR representation of a program \( P \), and \( B_b \) be the body of some method \( \Gamma_b[m] \). Following the definitions for a JBC program in Section 2.3, we define for BIR the CFG of a class as the disjoint union of the CFGs of the methods in the class, and the CFG of a program as the disjoint union of all CFGs of the classes in the program, as follows.

**Definition 10** (CFG Extraction from BIR). The instruction-wise extraction function \( \mathcal{G}_{\text{bir}} : (\text{Method-Ref} \times \text{Addr} \times \text{BirInstr}) \rightarrow \mathcal{P}(V_m \times L_m \times V_m) \) is defined by the rules in Figure 3.1. The method graph for \( m \) is defined as \( \mathcal{G}_{\text{bir}}(m) = \bigcup_{(p,i) \in B_b} \mathcal{G}_{\text{bir}}^{m,p,i} \). The control flow graph for the program is defined as \( \mathcal{G}_{\text{bir}}(\Gamma_b) = \bigcup_{\{m | \Gamma_b[m] \in \Gamma_b \}} \mathcal{G}_{\text{bir}}(m) \).

We subdivide the definition of \( \mathcal{G}_{\text{bir}} \) into two parts. The *intra-procedural* analysis extracts for every method an initial CFG, based solely on its instruction array, and its exception table. Based on these CFGs, the *inter-procedural* analysis computes the functions \( \bar{\mathcal{N}}_{pc,n} \), 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.

First, we describe the rules applied by the intra-procedural analysis. Assignments, \([\text{nop}]\) and \([\text{mayinit}]\) add a single edge to the next normal control node. The conditional jump \([\text{if expr pc'}]\) produces a branch in the CFG: control can go either to the next control point, or to the branch point \( pc' \). The unconditional jump \([\text{goto pc'}]\) adds a single edge to control point \( pc' \). The \([\text{return}]\) and \([\text{return expr}]\) instructions generate an internal edge to a return node, i.e., a node with the atomic proposition \( r \). Notice that, although both nodes are tagged with the same \( pc \), they are different because their sets of atomic propositions are different.

The BIR transformation provides the static type \( \text{staticT}(\text{expr}) \) of the exception raised by \([\text{throw expr}]\), and we soundly over-approximate the possible types of \( \text{expr} \) to all its subtypes. The extraction produces an exceptional edge for each type, followed by the appropriate edge derived from the exception table.

The rule for assertion instructions produces a normal edge, for the case that the implicit exception is not raised, and an edge to the exceptional node tagged with the exception type (as defined in Figure 2.12), together with the appropriate edge derived from the exception table.

The extraction rule for a constructor call \(([\text{new C}])\) produces a single normal edge, since there is only one possible receiver for the method call. In addition, we
3.1. EXTRACTION OF CFGS FROM BIR

The extraction rules for control flow graphs from BIR are as follows:

\[ \overline{H}_{pc,x,l} = \begin{cases} \{(\circ_{pc,l}, \bullet_{m'}, x, \varepsilon, \circ_{pc'}) \} & \text{if } \overline{H}_{pc,x}\Gamma_{b}[m] = \text{pc'} \neq 0 \\ \{(\circ_{pc,x,l}, \bullet_{m'}) \} & \text{if } \overline{H}_{pc,x}\Gamma_{b}[m] = 0 \end{cases} \]

\[ \overline{N}_{pc,n} = \bigcup_{x|\text{expr} \in V_{n}} \overline{H}_{pc,x,n} \]

\[ \overline{G}_{bir} = \bigcup_{i \in \text{Assignment} \cup \{\text{nop}\} \cup \text{MethodCall}} \overline{H}_{pc,x,e} \bigcup_{i \in \text{Throw expr}} \overline{H}_{pc,x,e} \bigcup_{i \in \text{Assertion}} \overline{H}_{pc,x,e} \bigcup_{i \in \text{NewObject}} \overline{H}_{pc,x,e} \bigcup_{n \in \text{res}} \alpha(n') \bigcup_{i \in \text{MethodCall}} \overline{N}_{pc,n} \]

Figure 3.1: Extraction rules for control flow graphs from BIR

also produce an exceptional edge, because a NullPointerException [58, § 6.5] is potentially raised.

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

Next, we describe the inter-procedural analysis. In all program points where there is a method invocation, the function \( \overline{N}_{pc,n} \) adds exceptional edges, relative to exceptions propagated by called methods. It analyzes whether the CFG of an invoked method \( n \) contains an exceptional return node. If it does, then function \( \overline{H}_{pc,x,n} \) verifies whether the exception of type \( x \) is caught in position \( pc \). If so, it adds two edges: one labelled with the signature of the called method \( n \), showing that it has terminated with an uncaught exception, and a second edge showing the transfer of control to the exception handler. Otherwise it adds an edge to an exceptional return node. In the latter case, the propagation of the exception continues until it is caught by some caller method, or there are no more methods to handle it. This is similar to the process described by Jo and Chang [47], who also present a fixed-point algorithm to compute the propagation edges. It checks the pre-computed call-graph to determine at which control points the invocations are made to a method propagating a given exception. If there is a suitable handler for that exception, it adds the respective handling edges, and the process stops. Otherwise, the computation proceeds.
3.2 Correctness of CFG Extraction

This section discusses the correctness proof of the CFG extraction algorithm. We start by pointing out that the operational semantics of BIR has been defined in [20]. Moreover, it has been proven that there is a semantic-preserving relation between the terminating traces for the same program in JBC and BIR. However, our algorithm is defined purely syntactically, thus we do not use BIR’s correctness result.

We prove correctness indirectly, using as a reference the idealized direct extraction algorithm $\mathcal{G}_{\text{jbc}}$ defined in Section 2.3. $\mathcal{G}_{\text{jbc}}$ is based directly on the semantics of Java bytecode, but assumes an oracle to predict the exceptions that can be thrown by each instruction. We exploit this idealized algorithm by proving that given a JBC program, the CFG produced by our extraction algorithm ($\mathcal{G}_{\text{bir}} \circ \text{BC2BIR}$) structurally simulates the CFG produced by the direct extraction algorithm ($\mathcal{G}_{\text{jbc}}$). We then reuse a result established previously by Huisman et al. [39, Theorem 1] that structural simulation entails behavioural simulation. As explained in Section 2.3, the latter result is stated over slightly different, but equivalent, definitions of CFG and CFG behaviour, and thus the result applies in our setting as well. By transitivity of simulation we conclude that the behaviour induced by the CFG extracted by $\mathcal{G}_{\text{bir}} \circ \text{BC2BIR}$ simulates the JVM behaviour of the original program.

![Figure 3.2: Schema for CFG extraction and correctness proof](image)

Figure 3.2 summarizes our approach. Notice that CFG behaviour is an induced notion, which gives rise to infinite state spaces. Thus, our approach of proving (finite) structural simulation and reusing the result from [39] is much more economic than proving behavioural simulation.

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

Preliminaries for the Correctness Proof

The BC2BIR transformation may collapse several bytecode instructions into a single BIR instruction. We divide the JBC instructions into two sets: the *producer* instructions, i.e., those that produce at least one BIR instruction in function BC2BIR\	extsubscript{instr}, and the *auxiliary* ones, i.e., those that produce none. This division can be deduced from Figure \ref{fig:bc2bir} (on page 29). For example, *store* and *invokevirtual* are producer instructions, while *add* and *push* are auxiliary.

We partition the bytecode instruction array into *bytecode segments*. These are sub sequences delimited by producer instructions. Thus, each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Equivalently, we may say that a segment is defined by an optional *basic block* \cite{basicblock} and a subsequent producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java Bytecode Verifier (see Section \ref{sec:bcv}). All methods in such programs must have *goto*, *return*, or *athrow* as the last (reachable) instruction, which are producer instructions. Therefore, there can not be contiguous (reachable) instructions that are not delimited by a producer.

Each bytecode segment is transformed into a set of contiguous instructions by BC2BIR. We call this set a *BIR segment*, which is a partition of the BIR instruction array. There exists a one-to-one mapping between bytecode segments and the BIR segments, which is also order-preserving. Thus, we can associate each instruction, either in the JBC or BIR arrays, to the unique index of its correspondent bytecode segment. Figure \ref{fig:bc2bir} (on page 31) illustrates the partitioning of instructions into segments. Method *odd* has eight bytecode (and BIR) segments, as indicated by the distinct shades. Producer instructions are underlined.

In the definition of the direct extraction algorithm in Figure \ref{fig:direct} (on page 24), one can observe that all auxiliary instructions give rise to an internal transfer edge only. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path-like graph of internal transfer edges of the same length as the number of auxiliary instructions, followed by the edges generated for the producer instruction. Let $p$ be the position for the first auxiliary instruction, and $q$ the position of the producer instruction. We illustrate the pattern for this path-like graph below.

$$
\begin{align*}
& c^p_m \rightarrow c^\text{succ}(p) \rightarrow c^\text{succ}(\text{succ}(p)) \rightarrow \ldots \rightarrow c^q_m \\
\end{align*}
$$

It is easy to see that every path-like graph is weakly simulated by some reflexive edge $c^p_m \rightarrow c^p_m$. Therefore, for simplicity we present the proof for the case where $p = q$. That is, for JBC segments without auxiliary instructions.

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

Based on the observations above, our main theorem states that the method graphs extracted using the indirect algorithm weakly simulate (see Definition 9) the method graphs using the direct algorithm. The abstract stacks are omitted in the proof, since examining the instructions is sufficient to produce the edges.

**Theorem 2** (Structural Simulation of CFGs). Let \( \Gamma \) be the environment modeling a well-formed JBC program \( P_{\text{jbc}} \). Then \( (G_{\text{bir}} \circ \text{BC2BIR})(\Gamma) \) weakly simulates \( G_{\text{jbc}}(\Gamma) \).

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

We define the binary relation \( R \) as the union of two binary relations, as presented in Definition 11 and show the relation to be a weak simulation in the standard fashion, following Proposition 1 (page 35): for every pair of nodes in \( R \), we first show that the nodes have the same set of atomic propositions, and then match every strong edge that has the first node as source, to a corresponding weak edge that has the second node as source, so that the target nodes are again related by \( R \).

**Definition 11** (Relation Between CFGs nodes).

\[
R \overset{\text{def}}{=} R_1 \cup R_2
\]

\[
R_1 \overset{\text{def}}{=} \{(\circ_{m}^{p}, \circ_{m}^{x,r}) \mid \text{seg}_{\text{JBC}}(m,p) = \text{seg}_{\text{BIR}}(m,\text{pc}) \land \text{pc} = \text{fst}(\text{seg}_{\text{BIR}}(m,\text{pc}))\}
\]

\[
R_2 \overset{\text{def}}{=} \{(\circ_{m}^{p}, XR, \circ_{m}^{x,r}) \mid \text{seg}_{\text{JBC}}(m,p) = \text{seg}_{\text{BIR}}(m,\text{pc}) \land \text{pc} \in \text{oap}(\text{seg}_{\text{BIR}}(m,\text{pc}), XR)\}
\]

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

Let $(\circ_m^p, \star) \in R_1$ and let $(\circ_m^p, \star) \in R_2$ where $\circ_m^p$ and $\circ_m^p$ are control nodes in $G_{bc}(\Gamma[m])$ and $\star$ is a control node in $(G_{bin} \circ BC2BIR)(\Gamma[m])$. We consider the two cases separately. Let first $(\circ_m^p, \star) \in R_1$. The proof proceeds by case analysis on the type of the producer instruction of the bytecode segment $seg_{BC}(m, p)$ giving rise to $\circ_m^p$. The cases follow the subsets of Java bytecode instructions presented in Figure 2.9 which share the same extraction rule in the direct algorithm for its instructions. We rely on the notation to indicate that two nodes have the same set of atomic propositions.

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

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

\[
G_{BC}^{m, p, \text{throw } x} = \begin{cases} 
\{ \circ_m^p \xrightarrow{x} \circ_m^p, \circ_m^p \xrightarrow{x} \circ_m^q \} & \text{if there is a handler} \\
\{ \circ_m^p \xrightarrow{x} \circ_m^p, \circ_m^p \xrightarrow{x} \circ_m^q \} & \text{otherwise}
\end{cases}
\]

The transformation $BC2BIR_{\text{instr}}$ returns a single instruction. Then, similarly to $G_{bc}$, the $G_{bin}$ function produces either one, or two edges:

\[
BC2BIR_{\text{instr}}^{m, \text{throw } e} = [\text{throw } e]
\]

\[
G_{\text{bin}}^{m, \text{throw } e} = \begin{cases} 
\{ \circ_m^p \xrightarrow{x} \circ_m^p, \circ_m^q \xrightarrow{x} \circ_m^q \} & \text{if there is a handler} \\
\{ \circ_m^p \xrightarrow{x} \circ_m^p, \circ_m^p \xrightarrow{x} \circ_m^p \} & \text{otherwise}
\end{cases}
\]

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

In the case where there is no exception handler for $x$, the only edge is $\circ_m^p \xrightarrow{x} \circ_m^p$, which is matched by $\circ_m^p \xrightarrow{x} \circ_m^p$. Moreover, $(\circ_m^p, \star) \in R$ since $seg_{BC}(p) = seg_{BIR}(m, pc')$ and $pc' \in \text{oap}(s, \{x, r\})$. That is, $pc'$ tags the node where the set $XP = \{x, r\}$, which concludes the case.
CHAPTER 3. CFG EXTRACTION FROM COMPLETE PROGRAMS

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

\[
\begin{align*}
\text{BC2BIR}_{\text{instr}}^{\text{return}} &= \left[\text{return expr}\right] \\
G_{\text{bir}}^{m, pc, \text{[return expr]}} &= \{\circ_{m}^{pc, r} \rightarrow \circ_{m}^{pc, r}\}
\end{align*}
\]

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

3.3 The ConFlEx Tool for Complete Programs

The concrete, indirect extraction algorithm is implemented as our CFG extraction tool ConFlEx [31]. It uses SawJa [37], a library for the static analysis of Java bytecode programs. SawJa features the most popular virtual method call resolution algorithms, and the transformation from bytecode into BIR. However, the standard implementation of SawJa only performs a syntactic transformation, and does not compute the object types. We have instrumented the BIR transformation from SawJa to compute the most generic object type. Thus, we can estimate soundly the type of explicit exceptions.

ConFlEx implements the extraction rules from Figure 3.1 (on page 41), including the computation of exception propagation. Currently the extraction is fine-tuned for the particular variant of control flow graphs, presented in Definition 2. However, it can be extended to support other formalizations with a relatively small effort. Also, the control node addresses are relative to the BIR representation of the analyzed program. Though possible, it requires a medium effort to implement the mapping from the extracted CFG to the original JBC addresses.

We have evaluated the performance of ConFlEx. It is future work to also evaluate the precision of the extracted CFGs in terms of the ratio of spurious nodes, and to experiment with different VMC resolution algorithms. To evaluate our tool, we have applied it to a collection of real-world applications. The main criterion when selecting these applications has been their size (number of instructions), without verification in mind. We aimed to diversify the type of applications. For instance, JFlex and Java-Cup are parsing tools, TJWS is a web server, and JExplorer is an LDAP browser. In our evaluation we use the Rapid Type Analysis [10], which provides the best balance between performance and precision [77]. All experiments are done on a server with an Intel i5 2.53 GHz processor and 4GB of RAM. Methods from the API are not extracted; only classes that are part of the program are
3.3. THE CONFLEX TOOL FOR COMPLETE PROGRAMS

### Table 3.1: Statistics for ConFlEx

<table>
<thead>
<tr>
<th>Program</th>
<th># of BIR instr.</th>
<th># of BIR instr.</th>
<th>BIR time (ms)</th>
<th>Intra-Procedural # of nodes</th>
<th>Intra-Procedural # of edges</th>
<th>Intra-Procedural time (ms)</th>
<th>Inter-Procedural # of nodes</th>
<th>Inter-Procedural # of edges</th>
<th>Inter-Procedural time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jasmin</td>
<td>30930</td>
<td>10850</td>
<td>524</td>
<td>26595</td>
<td>20937</td>
<td>20595</td>
<td>27267</td>
<td>27717</td>
<td>35</td>
</tr>
<tr>
<td>Java-Cup</td>
<td>31958</td>
<td>13174</td>
<td>679</td>
<td>24263</td>
<td>24549</td>
<td>21500</td>
<td>32031</td>
<td>32449</td>
<td>43</td>
</tr>
<tr>
<td>TJWS</td>
<td>33653</td>
<td>13129</td>
<td>423</td>
<td>32333</td>
<td>34325</td>
<td>515</td>
<td>71218</td>
<td>78233</td>
<td>354</td>
</tr>
<tr>
<td>JFlex</td>
<td>53474</td>
<td>20433</td>
<td>1191</td>
<td>39429</td>
<td>40035</td>
<td>675</td>
<td>53956</td>
<td>54777</td>
<td>102</td>
</tr>
<tr>
<td>JXplorer</td>
<td>186644</td>
<td>77536</td>
<td>1129</td>
<td>150495</td>
<td>154583</td>
<td>21450</td>
<td>332366</td>
<td>344995</td>
<td>2636</td>
</tr>
<tr>
<td>Xowa</td>
<td>713160</td>
<td>294339</td>
<td>77468</td>
<td>716362</td>
<td>719534</td>
<td>146019</td>
<td>5741597</td>
<td>5805206</td>
<td>396437</td>
</tr>
<tr>
<td>Soot</td>
<td>1345574</td>
<td>516404</td>
<td>64856</td>
<td>1081297</td>
<td>1081962</td>
<td>4055987</td>
<td>14307406</td>
<td>14319126</td>
<td>3665739</td>
</tr>
</tbody>
</table>

considered. Table 3.1 shows statistics about the analyzed programs, the size of the produced CFGs and the extraction times.

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

We can also observe that, on average, the computation time for intra- and inter-procedural analysis grows proportionally with the number of BIR instructions. However, this growth depends heavily on the number of exceptional paths in the analyzed program. Surprisingly, the inter-procedural analysis, which is a fixed-point computation, showed in practice to only take a fraction of the computation time of the intra-procedural analysis, with Xowa again being the only exception. We conjecture that the deviation for Xowa is caused by a more significant presence of exceptional flows in its implementation.

We do not provide comparative data with other extraction tools such as Soot [82] or Wala [11], since this would demand the implementation of extraction rules corresponding to ours, but from their intermediate representations. However, experimental results from SAWJA [37] show that it outperforms Soot in all tests w.r.t. the transformation into their respective intermediate representations, and outperforms Wala w.r.t. virtual method call algorithms. Thus, our extraction algorithm clearly benefits from using SAWJA and BIR.
3.4 Discussion

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

3.4.1 Precision of the Extraction

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

In our analysis, exceptions are the only data type considered. However, the potential occurrence of implicit (runtime) exceptions depends heavily on program data. For instance, whether an ArithmeticException is raised because of a division by zero, or whether an ArrayIndexOutOfBoundsException is raised because of an array access with an invalid index, depends on the instruction’s operands. Our extraction algorithm does not consider such data, therefore it necessarily over-approximates the implicit exceptions of every Java bytecode instruction to the ones it may potentially raise, as defined in the JVM specification [58]. The necessary information for this is obtained from the assertions (see Figure 2.12) introduced by the BIR transformation.

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

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

The primary utility of our CFGs is the verification of control flow based temporal safety properties. This allows the CFG extraction to abstract from most of the data. This design choice gives rise to a comparatively lightweight approach that is both efficient and easy to be shown sound. Still, our extraction algorithm can benefit from finer data flow analyses such as null pointer analysis [75] or symbolic execution [51], provided that these analyses are also proven to be sound. Further-
more, the latter technique is envisaged in [72], as the means to extract CFGs with symbolic data.

### 3.4.2 Exceptions Supported by the Extraction

As explained in Section 3.1, the concrete extraction algorithm \( G_{	ext{bir}} \circ \text{BC2BIR} \) extracts CFGs soundly, considering all explicit exceptions, i.e., raised by \texttt{throw}, and a subset of the implicit exceptions (Figure 2.12). Here we clarify which exceptions are supported, and discuss the implications of supporting the full set of exceptions.

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

Many implicit exceptions are raised during the execution of a particular instruction. However, there are also implicit exceptions that depend on the JVM execution settings (e.g., allocated memory), its implementation, or the hosting machine. All these exceptions are subtypes of the class \texttt{java.lang.Error}, which according to the Java specification [58] represent severe problems, and applications should not try to recover from them. The goal of our tool is to extract models for typical software engineering processes, such as formal software verification. Adding exceptions relative to the execution environment, which can be raised virtually at any control point, would result in bloated CFGs that do not contribute to the goal of verification. Therefore, our algorithm does not support implicit exceptions that are subtypes of \texttt{Error}. The single outlier is \texttt{ExceptionInInitializerError} since it is not raised by an environment anomaly, but due to an exception raised by the execution of a class initializer.

On the other hand, the implicit exceptions caused by abnormal execution of an instruction are supported. Demange \textit{et al.}[26] have made a careful analysis of the official Java specification, and encode exceptions by means of BIR assertions. Currently, \texttt{IllegalMonitorStateException} is the only unsupported exception. It is intrinsic to concurrent programs, and does not impact the algorithms described in this paper, which are defined for sequential programs. Nevertheless, our algorithm can be easily adapted to accommodate this, or any potentially new exception that newer versions of BIR may support by means of assertions.

We conclude by mentioning that our tool, \texttt{ConfLEX}, supports the full set of exceptions, as defined in the standard Java API. However, the user must explicitly indicate that API methods are part of the analyzed program since the parsing and extraction of such methods impact the analysis time and the CFGs’ size. Therefore, the default setting is to skip API methods. In that case, if an API method is invoked inside a program, the tool still considers the exceptions listed in the \texttt{throws} clause of the method declaration (as illustrated by Figure 2.8, where the \texttt{parseInt} method declares that it may propagate a \texttt{NumberFormatException}).
3.4.3 Multi-threaded Control Flow

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

In previous work Huisman et al. define a notion of CFGs for multithreaded programs [39, Definition 9]. It considers the following primitives for thread synchronization in JBC programs: thread spawning and joining, lock acquiring and releasing, and wait, notify, and notifyAll.

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

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

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

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

3.5 Related Work

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

Zhao [86] presents an initial formal definition of CFGs for Java bytecode programs. However, there is no formal definition of an extraction algorithm, only a description of the relevant aspects of the transformation from JBC to the control flow graphs. Among these, the paper considers exceptions and describes how exceptions are handled. However, it does not discuss how they are raised, or how to estimate the exception type.

Sinha et al. [70, 71] propose a control flow graph extraction algorithm for both Java source and bytecode, which takes into account explicit exceptions only. The algorithm performs first an intra-procedural analysis, computing the exceptional return nodes caused by uncaught exceptions. Next, it executes an inter-procedural analysis to compute exception propagation paths. This division is similar to how our algorithm analyses exceptional flows, using a slightly different inter-procedural analysis. However, the authors do not discuss how the static type of explicit exceptions is determined by the bytecode analysis, whereas we get this information from the BIR transformation. Moreover, the use of BIR allows us to also support (a subset of the) implicit exceptions.

Jiang et al. [46] extend the work of Sinha et al. to C++ source code. C++ has the same scheme of try-catch and exception propagation as Java source, but without the finally blocks, or implicit exceptions. This work does not consider the exception types. Thus, it heavily over-approximates the possible flows by connecting the control points with explicit throw within a try block to all its catch blocks, and considering that any called method containing a throw may terminate exceptionally. Instead, our work considers the exception types, and thus produces finer CFGs. Moreover, it tells which exceptions can be raised, or propagated from method invocations.

Choi et al. [16] use an intermediate representation from the Jalapeño compiler [13] to extract CFGs with exceptional flows. The authors introduce a stackless representation, using assertions to mark the possibility of an instruction raising an exception. This approach was followed by Demange et al. when defining BIR, and proving the correctness of the transformation from bytecode. As a result, our extraction algorithm, via BIR, is very similar to that of Choi. We differ by defining formal extraction rules, and proving its correctness w.r.t. the behaviour.

Several algorithms about the resolution of virtual method calls exist in the literature. The most popular ones are Class Hierarchy Analysis (CHA) [25, 79] and Rapid Type Analysis (RTA) [77, 79]. In the former, the sets of possible method call receivers are built by simply looking at the inheritance relations between the classes; in the latter, by considering the instantiation of objects. CHA is widely used because of its simplicity and performance, although it is not precise. RTA is considered to provide a good trade-off between performance and accuracy of the results. Both algorithms rely on code analysis. Nevertheless, in a modular scenario a VMC must be resolved despite the absence of some components. Thus, we introduce MCA in in Section 4.2 which is essentially a simplification of CHA,
and over-approximate the receivers soundly by using the class hierarchy only.

Jo and Chang [47] construct CFGs from Java source code by computing normal and exceptional flows separately. An iterative fixed-point computation is then used to merge the exceptional and the normal control flow graphs. Our exception propagation computation follows their approach; however, the authors do not discuss how the exception type is determined. Also, only explicit exceptions are supported; in contrast, we determine the exception type and support implicit exceptions by using the BIR transformation.

Recently, Mihancea and Minea [60] presented jModex, a tool for the extraction of finite-state models from Java web applications that are tailored for the model checking of security properties. In contrast to ConFlEx, their tool does not fully support virtual method calls, nor exceptional flow.

Finally, we cite Bandera [17, 28] as a pioneering tool to generate abstract models from Java source programs for model checking. It contains several features, such as output for multiple model checkers, and some static analyses, such as slicing. In comparison to ConFlEx, Bandera is a versatile tool, which provides an integrated framework to program checking. The work mentions the support of exceptions as future work. However, we could not find other references about exceptions in further publications about Bandera.
Chapter 4

CFG Extraction from Incomplete Programs

In this section we generalize the previous definitions, to handle incomplete Java bytecode systems. First, we extend the formal JVM framework described in Section 2.1 to represent unavailable software components as user-provided interfaces. Next, we generalize the indirect algorithm presented in Section 3.1 to modularly extract CFGs from the available components. That is, the algorithm extracts the method graphs for the available methods, and resolves the inter-dependencies involving missing methods by using the provided interfaces.

Eventually the missing components will be instantiated, and the program will become complete. Thus, we define constraints over the arrival of components that preserve properties established over the CFGs that were extracted when the program was incomplete. We conclude by showing that if the newly arrived components comply to the defined constraints, then the CFGs extracted with the modular algorithm from an incomplete program are sound over-approximations of the CFGs extracted from any complete system assembled from it. Therefore, the safety properties verified over the CFGs still hold.

We conclude by describing the extension of ConFlEx and Sawja for analyzing incomplete programs. We re-evaluate the tool over some test cases.

The work presented in this chapter has been published in [33].

4.1 Incomplete Java bytecode programs

Software systems are typically created by the assembly of several components. Each software component provides specialized functionalities to other components, and defines how the interaction occurs by providing an interface. We call an incomplete system (or program) any software system which has all its components defined, but the implementation of at least one of the components is missing.
We model incomplete JBC programs as open environments, as presented in Figure 4.1. Most of the definition is similar to Definition 2.3 of closed environments for complete programs. For instance, $\Gamma_o$ is defined as the union of the partial mappings from names to its classes, interfaces, and methods. Also, the mappings $\Gamma^I_o$ and $\Gamma^C_o$ are defined exactly as $\Gamma^I$ and $\Gamma^C$, respectively.

$$\Gamma^I_o : \text{Interface-Name} \rightarrow \left\{ \begin{array}{l} \text{interfaces} : \text{set of Interface-Name} \\ \text{method} : \text{set of Interface-Method-Ref} \end{array} \right\}$$

$$\Gamma^C_o : \text{Class-Name} \rightarrow \left\{ \begin{array}{l} \text{super} : \text{Class-Name} \\ \text{interfaces} : \text{set of Interface-Name} \\ \text{fields} : \text{set of Field-Ref} \end{array} \right\}$$

$$\Gamma^M_o : \text{Method-Ref} \rightarrow \left\{ \begin{array}{l} \text{code} : (\text{ADDR} \times \text{Instruction})^* \\ \text{handlers} : \text{Handler}^* \end{array} \right\}$$

$$\Gamma_o = \Gamma^I_o \cup \Gamma^C_o \cup \Gamma^M_o$$

Figure 4.1: Open environment of a JBC/BIR program

The most notable difference is the partial mapping from names to method implementations, $\Gamma^M_o$. First, the code array allows the existence of methods with empty bodies, representing the missing components. For such methods, the definition of $\Gamma^M_o[m]$.handlers has a different interpretation than $\Gamma^M[m]$.handlers. Each handler determines one exception type that can never be propagated by the missing method, when it becomes available. Here, the addresses are irrelevant since the definition applies to the whole method body.

Open environments model the BIR version of incomplete programs in the same way as closed environments model the complete BIR programs. We use this common modeling of BIR programs to again define the CFG extraction indirectly, with the syntactic transformation of JBC into BIR as the one presented in Section 2.4.

In this chapter, we focus on the extraction of CFGs from the BIR representation.

We now formally define two concepts that have been previously explained. The first is the type hierarchy, presented in Definition 12. It is simply defined as the union of the partial maps that contain the definitions of data types, i.e., classes and interfaces. The type hierarchy gives rise to the definition of the exceptions set, as presented in Definition 13. These definitions are necessary for defining the extraction algorithm for incomplete programs, presented in the following section.

**Definition 12 (Type Hierarchy).** Let $\Gamma^I_o$ and $\Gamma^C_o$ be partial maps from interface and classes names, respectively, to their attributes. The Java type hierarchy is defined as $\mathcal{L} = \Gamma^C_o \cup \Gamma^I_o$. 
Definition 13 (Exceptions Set). Let \( \mathcal{L} \) be a given class hierarchy. We define \( \mathcal{A}_{\text{ny}} = \{ c \in \mathcal{L} \mid c <_{\mathcal{C}} \text{java.lang.Throwable} \} \), which represent the exceptions type in a given open environment.

4.2 The \( \mathcal{O}_G \) Extraction Algorithm

In this section we present a modular CFG extraction algorithm that receives an open environment, and produces CFGs for the available components. It is a generalization of the \( \mathcal{G}_{\text{bir}} \) algorithm for complete programs, presented in Section 3.1.

The extraction is also defined for BIR programs. Once again, the motivation to use BIR is its support for exceptions. The transformation from JBC to BIR is purely syntactic. Thus the mapping between an open Java bytecode environment into an open BIR environment is trivial: the partial maps \( \Gamma \) are the same for both environments, and the mapping \( \Gamma_M \) is the result of the transformation \( \text{BC2BIR} \) for each available method.

We define the transformation function \( \mathcal{O}_G \) as follows.

Definition 14 (CFG Extraction from Incomplete BIR). The instruction-wise extraction function \( \mathcal{O}_G : (\text{Method-Ref} \times \text{Addr} \times \text{BirInstr}) \to \mathcal{P}(V_m \times L_m \times V_m) \) is defined by the rules in Figure 4.2. The method graph for \( m \), s.t. \( |\Gamma_o[m].\text{code}| > 0 \), is defined as \( \mathcal{O}_G(m) = \bigcup_{(p,i) \in \Gamma_o[m]} \mathcal{O}_G^{m,p,i} \). The control flow graph for the incomplete program is defined as \( \mathcal{O}_G(\Gamma_o) = \bigcup_{(m \mid \Gamma_o[m] \in \Gamma_o)} \mathcal{O}_G(m) \).

The extraction rules for CFGs are very similar to the ones presented Figure 3.1. In fact, the rules for instructions that only depend on intra-procedural information (a method’s own instructions and exception handlers table) are the same in both algorithms. This is the case for all instructions, except the ones that execute method calls: \text{NewObject} and \text{MethodCall}. The differences to the algorithm for complete programs are the resolution of virtual method calls, and the propagation of uncaught exceptions. We highlight with a darker shade the affected instruction-wise extraction rules, and auxiliary functions in Figure 4.2.

The \( \mathcal{G}_{\text{bir}} \) algorithm for complete programs is parametrized by a sound VMC resolution algorithm. However, standard VMC algorithms, such as the Rapid Type Analysis (RTA) \[10\], are defined for complete programs only, and may provide unsound estimation in the absence of code. We therefore fix the VMC resolution algorithm of \( \mathcal{O}_G \) to our Modular Class Analysis (MCA), which is a generalization of the Class Hierarchy Analysis (CHA) \[25\]. MCA soundly over-approximates the set of possible receivers to a VMC as the methods with the same signature (ns) from subtypes and from the closest super-type of the static type (C) that are either provided or declared to be missing (given by function \( \text{dom} \)).

The other modification concerns the function \( \mathcal{N} \) that computes the control flow caused by exception propagation. The definition of \( \mathcal{N} \) contains two cases: the first one is when the called method is available in the current open environment \( \Gamma_o \). Here, we proceed in the same fashion as with the analysis of complete programs. That
is, for each exceptional return node in the called method’s CFG (which denotes a propagated exception), we add a set of edges relative to handling or propagation of the exception, as computed in $H$.

The second case is when the called method is missing. The analysis of exceptional flow in this scenario is more complex because the set of exceptions that a method can raise depends on the instructions from the method. Thus it is impossible to determine in advance the exact exception types that a missing method may propagate. One could suggest that the user guarantees by contract the set of the possible exceptions a method can raise, and verify the set upon the arrival of the missing method. However, this approach is too restrictive since it forces the user to annotate the set of propagated exceptions even before the method code is implemented. We chose a dual approach instead.
4.3 Correctness Proof

We conservatively over-approximate the set of propagated exceptions by a missing method. First, we assume this to be the set Any of all exception types. Then, the user may annotate a set of exceptions that the missing method cannot propagate, defined as \( \Gamma^m \). Notice that the user-provided annotation assists the automatic extraction to over-approximate the set of propagated exceptions to the least possible set, and as a consequence to produce more compact CFGs.

4.3 Correctness Proof

The main purpose of the extraction algorithm for incomplete systems is to extract CFGs from the available components of incomplete JBC programs that are sound over-approximations of CFGs extracted for any instantiation of the unavailable code. CFGs that have this property preserve the verification of temporal safety properties, as illustrated in Example 5 (on page 53). Further, the extraction definition 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.

Both the definition of the refinement relation, and the formal proofs about the property preservation are long and contain many details. Therefore, in this section we summarize the definitions and formal arguments for helping the reader’s comprehension. We refer to Appendix B for the full account.

Definition 15 (Environment Refinement). Let \( \Gamma_o \) and \( \Gamma'_o \) be open environments. We say that environment \( \Gamma_o \) refines environment \( \Gamma'_o \), written \( \Gamma_o \preceq \Gamma'_o \), if the following conditions hold:

(i) method references, class names and interface names defined in \( \Gamma'_o \) must also be in \( \Gamma_o \);

(ii) an interface in \( \Gamma'_i \) contain the same methods, and extend a subset of the interfaces in \( \Gamma'_i \);

(iii) classes in \( \Gamma'_c \) have the same super-class, implement a subset of the interfaces of the same classes in \( \Gamma'_c \);

(iv) a method in \( \Gamma'_o \) must have a superset of the handlers of \( \Gamma'_o \) 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 \( \Gamma_o \) cannot propagate exceptions declared in \( \Gamma'_o \) handlers, where it was unavailable.

We say that \( \Gamma \) implements \( \Gamma_o \) whenever \( \Gamma \preceq \Gamma_o \) and \( \Gamma \) is closed.
The refinement of a method which is unavailable in both the original and the refined environments entails that in \( \Gamma_o \) it propagates at most the same set of exceptions as in \( \Gamma_{o'} \). Thus, a CFG extraction from \( \Gamma_{o'} \) must over-approximate the set of propagated exceptions involving the method. In a refinement where a method is implemented in both environments (second case of (iv)), there cannot be changes; otherwise, the method graph extracted from \( \Gamma_{o'} \) would not soundly over-approximate the method graph from \( \Gamma_o \). The refinement of an unavailable method in \( \Gamma_{o'} \), which is implemented in \( \Gamma_o \), simply guarantees that it respects its interface w.r.t. propagated exceptions.

Now we enunciate the theorems that, combined, entail that the CFGs extracted from incomplete JBC programs soundly over-approximate the CFGs for the common methods (i.e., methods available in incomplete program) if the refinement relation holds. Again, we refer to Appendix B for the full account.

The following result establishes monotonicity of CFG extraction w.r.t. refinement. We show that method CFGs extracted from a refined environment are subgraphs of the CFGs extracted from the original open environment.

**Theorem 3** (Containment of CFGs). Let \( \Gamma_o \) and \( \Gamma_{o'} \) be open environments, and \( m \) be the signature of a method available on both. Then \( \Gamma_o \preceq \Gamma_{o'} \) implies \( oG(m, \Gamma_o) \subseteq oG(m, \Gamma_{o'}) \).

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 4** (CFG equality). Let \( \Gamma \) be a closed environment, and \( G^{MCA}_{\text{bir}} \) be the instantiation of \( G_{\text{bir}} \) with MCA. Then \( G^{MCA}_{\text{bir}}(\Gamma) = oG(\Gamma) \).

These results ensure soundness of the CFG extraction w.r.t. temporal safety properties, by virtue of several results established earlier [5, 35]. First, subgraph inclusion of CFGs entails structural simulation between the CFGs. Next, structural simulation in turn entails behavioural simulation ([35, Th. 36]). Third, temporal safety properties are preserved (backwards) under behavioural simulation ([35, Cor. 17]). These three results guarantee preservation of temporal safety properties under refinement of open environments. Together with the soundness result for \( G_{\text{bir}} \) established in [5] and Theorem 4 above, we obtain soundness of \( oG \).

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 thus, 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 fewer over-approximations w.r.t. exceptional flow.
4.4 The ConFlEx Tool for Incomplete programs

We have extended the ConFlEx tool presented in 3.3 to support the definitions for incomplete programs. Now we present the implementation details, and the tool re-evaluation over another set of test cases.

We have tailored Sawja in several parts. The main 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 newly-introduced definitions as modules in Sawja. The module DefCFG represents and manipulates CFGs, as presented in Definitions 3 and 5. We wrote it as a separate module to make ConFlEx independent from a single CFG definition. This facilitates in the future to support other CFG definitions. The OpenEnv module implements the representation of an incomplete Java bytecode system as an open environment, following the definitions in Figure 4.1. The environment is represented by a structure containing three associative maps, representing $\Gamma_I$, $\Gamma_C$, and $\Gamma_M$. The module also implements the check of the refinement relation, following the rules in Figure B.1.

The module MCA implements the virtual method call resolution algorithm for incomplete systems. The algorithm over-approximates the set of possible receivers to a virtual call as presented in Section 4.2: 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 [64], 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 programmers may question the choice of using dummy methods instead of abstract methods. The reason is that abstract methods are only allowed inside abstract classes, which in turn cannot be instantiated. I.e., an available component cannot create objects of an abstract class. Compilers see this as an error, and do not generate the .class for an available component. 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 4.3 and put it into each directory containing an annotated method.
import java.lang.annotation.*;

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

Figure 4.3: The GhostComponent annotation

The user annotates an unavailable method by adding a custom GhostComponent annotation immediately before the declaration. 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 Retention and Target define 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 methods (including constructors) are the elements that can be annotated.

Figure 4.4 shows examples of annotated missing methods. Figure 4.4a presents the annotation of method even’s interface, as mentioned in Example 6: the method may only call odd, or itself, and may not propagate exceptions. Figure 4.4b presents a more general example, which provides the required method with fully-qualified method name, and lists the three types of exceptions that the missing method cannot propagate.

By default, 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 only considers the client program, which is the code that the user explicitly declares to belong to the program by setting the path to its .class files. The client code must contain an entry method main. 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 methods from the client program. Also, we assume that API methods can only propagate exceptions declared in the throws field of the method’s declaration. The user can alter ConFlEx’s assumptions w.r.t. API methods by explicitly declaring the standard API to be a component of the program. However, the Java standard API is large, and the consequence are unnecessarily large CFGs.

We have introduced in Section 4.1 the set $\mathcal{A}$ of all exception types in an open environment, and have presented it in Definition 13 as all the subtypes of java.lang.Throwable. In practice, we can filter out exception types that cannot be raised because they are neither referenced in the code, nor in the interfaces. We
compute \textit{Any} as the union of: \textit{(a)} exceptions represented by the BIR assertions (Figure 2.12); \textit{(b)} user-defined exception classes; and \textit{(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 \textsc{ConFlEx} is executed with the same configuration options. The feature exploits the fact that the computation of the intra-procedural analysis is preserved when the implementation of a method is not altered. Still, \textsc{ConFlEx} recomputes the entire inter-procedural analysis, so that the control flow caused by propagated exceptions is over-approximated as little as possible.

\textsc{ConFlEx} creates an XML file and a text file for caching of previous analyses. The XML file stores the results from the intra-procedural analysis: the CFG, the
set of propagated exceptions, and method calling points. The file with the data cached data structure is presented in Appendix C. ConFlEx uses the external library XML-Light [62] for parsing the XML files. 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.

We re-validate ConFlEx by using real-world Java applications to mimic incomplete Java bytecode systems. We replace the implementation of some of the classes with annotated methods. Then, we re-introduce the implementations incrementally, to mimic the arrival of code. We choose three large, existing complete JBC applications, that have already been used as test cases in Section 3.3: Jasmin version 2.4 [59]; JavaCUP version 11a beta [38]; JFlex version 1.4.3 [52].

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-procedural 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 complete programs, with MCA, and RTA, to investigate the impact of the chosen VMC resolution algorithm on the size of the resulting CFGs. Notice that configuration [9] is the same as the one presented in Table 3.1 for complete programs. We have re-evaluated the test cases for Jasmin, JavaCUP and JFlex in the same software and hardware set-up as the other configurations, and thus to have comparable results.

In practice, we have 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 Any set. The missing classes in Scenario [1] 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 4.1 shows the experimental results. The considered data are: number of JBC instructions; number of nodes and edges of the CFG after the inter-procedural analysis; time of the intra-procedural analysis; and time for the inter-procedural analysis. The tests have been made on a Linux system running in 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
Table 4.1: Experimental results for ConFlEx

<table>
<thead>
<tr>
<th>Conf.</th>
<th>VMC</th>
<th>Reused results</th>
<th>Missing classes</th>
<th># of instrs.</th>
<th># of Nodes</th>
<th># of Edges</th>
<th>Intra</th>
<th>Inter</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<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>
<td>339</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>yes</td>
<td>1</td>
<td>30377</td>
<td>35684</td>
<td>36228</td>
<td>291</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>no</td>
<td>1</td>
<td>30377</td>
<td>35684</td>
<td>36228</td>
<td>1540</td>
<td>104</td>
<td></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>
<td>104</td>
<td></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>
<td>85</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>RTA</td>
<td>no</td>
<td>0</td>
<td>30930</td>
<td>27267</td>
<td>27717</td>
<td>690</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>yes</td>
<td>1</td>
<td>30042</td>
<td>76511</td>
<td>77345</td>
<td>1799</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>yes</td>
<td>1</td>
<td>33354</td>
<td>76798</td>
<td>77649</td>
<td>567</td>
<td>427</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>no</td>
<td>1</td>
<td>33354</td>
<td>76798</td>
<td>77649</td>
<td>2098</td>
<td>518</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>yes</td>
<td>0</td>
<td>35422</td>
<td>45455</td>
<td>46328</td>
<td>66</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>no</td>
<td>0</td>
<td>35422</td>
<td>45455</td>
<td>46328</td>
<td>2126</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>RTA</td>
<td>no</td>
<td>0</td>
<td>31958</td>
<td>32031</td>
<td>32449</td>
<td>983</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>yes</td>
<td>1</td>
<td>52336</td>
<td>118414</td>
<td>119868</td>
<td>6396</td>
<td>877</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>yes</td>
<td>1</td>
<td>55972</td>
<td>77174</td>
<td>78678</td>
<td>960</td>
<td>631</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>no</td>
<td>1</td>
<td>55972</td>
<td>77174</td>
<td>78678</td>
<td>7227</td>
<td>407</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>yes</td>
<td>0</td>
<td>60417</td>
<td>72154</td>
<td>73175</td>
<td>115</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>no</td>
<td>0</td>
<td>60417</td>
<td>72154</td>
<td>73175</td>
<td>7219</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>RTA</td>
<td>no</td>
<td>0</td>
<td>53474</td>
<td>53896</td>
<td>54777</td>
<td>1676</td>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>

2 and 3, where a single class is missing. This can be explained partially by the excessive over-approximation of the exceptional control flow.

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 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 2.5.
4.5 Related Work

The work presented in this chapter 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 [18], and is 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 [28] is a pioneering tool to generate abstract models from Java source programs. It is built on top of the Soot framework [54], 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 for program checking. However, it cannot analyze incomplete programs, and it does not address exceptional flows.

Dagenais and Hendren [19] 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 [41], 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 Lhoták [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.

The generation of CFGs from incomplete programs shares similarities with the extraction of so-called environments (not to be confused with program environments, as defined in Figure 2.3), which are models of the external behaviour as perceived by the software component under analysis. One may see the extraction of CFGs from the available components as similar to the environment generation for the missing components. Tkachuk and Dwyer [80] extend Bandera for the computation of environments, which are used for the modular model checking of soft-
ware components. However, the generation of environments is not modular itself. Tkachuk and Rajan \cite{81} refine the previous approach by adding limited support for exceptions, but admittedly producing unsound models.

Rountev \cite{68} presents a technique to identify correctly side-effect free methods in Java software. The approach is based on a class analysis, that determines the possible classes which an object reference points to. In our work, we assume that all the methods in the API libraries are side-effects free. Even though the described approach differs from ours, it might be used to prove or disprove our assumption. We could apply this technique preliminarily, before the extraction and analysis. Also, possible side-effects might be added to the control flow graph extracted.

Several works propose different exception analyses. Our algorithm follows the approach of Jo and Chang \cite{47} to extract CFGs by decoupling the intra- and inter-procedural analyses of exceptional control flow. However, they do not discuss implicit exceptions, nor do they address virtual method calls. Li et al. \cite{57} 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.
Chapter 5

Verifying Synchronization with Condition Variables

In this chapter, we define a formal verification technique, and show how to extract and inspect a program model to verify a specific correctness property.

We present a technique to automatically verify the synchronization of concurrent Java programs with condition variables, and its implementation as the STAVe tool. The property of interest is defined as “If every thread synchronizing under the same condition variables eventually enters its synchronization block, then every thread will eventually exit the synchronization”. The property entails that no thread will block indefinitely, waiting for a notification to resume the execution.

To support the verification, we propose the intermediate language SyncTask, a simple imperative language for specifying parallel computations that synchronize via condition variables. SyncTask programs can be extracted from Java programs. To assist the full automation of the extraction for Java programs with bounded data domains and numbers of threads, we propose an annotation language, and require programmers to provide hints to STAVe in the form of annotations in this language. We establish that for correctly annotated programs, the above-mentioned property holds if and only if the corresponding SyncTask program terminates.

STAVe transforms the termination problem into a reachability problem on hierarchical Coloured Petri Nets (as defined in Section 2.7), which is solved efficiently by tools like CPN Tools. We define rules to extract nets from SyncTask programs, and establish that a SyncTask program terminates if and only if its extracted Coloured Petri Net always reaches a special set of dead configurations. Figure 5.1 summarized the approach.

We evaluate STAVe on two annotated Java programs by extracting SyncTask programs and then their nets, and feeding the latter into CPN Tools.

The work described in this chapter is presented in [21].
5.1 Overview of the Approach

In this section we illustrate our verification method by presenting the artifacts that STaVe manipulates: an annotated Java program, its corresponding SyncTask program, and Coloured Petri Net. We then describe the CPN analysis.

The Java program in Figure 5.2 implements a shared Buffer. Producer and Consumer threads synchronize via the implicit monitor associated with the buffer object \( b \) to add or remove elements, and wait if the buffer is full or empty, respectively. Waiting threads are woken up by \texttt{notifyAll} after an operation is performed on the buffer, and compete for the monitor to resume execution.

The annotations are provided in comment blocks, and delimit the expected synchronization. The \texttt{@syncblock} annotations include the \texttt{synchronized} blocks to the observed synchronization behaviour, and \texttt{@monitor} and \texttt{@resource} map local references to global aliases. The annotation \texttt{@resource} above \texttt{Buffer} starts the definition of a resource type, i.e., an abstraction of a data type that is accessed in the synchronization. \texttt{@value}, \texttt{@object} and \texttt{@capacity} define the resource’s abstract state, and \texttt{@operation} and \texttt{@predicate} define how the class methods operate on the state. The \texttt{@synctask} annotation above \texttt{main} starts the declaration of locks, CVs and resources, and \texttt{@thread} annotations add the following objects to the global thread composition.

The SyncTask program in Figure 5.3 was automatically extracted by STaVe from the Java program in Figure 5.2. The two thread types, \texttt{Consumer} and \texttt{Producer}, preserve the synchronization behaviour, and the \texttt{main} block contains variable declarations and initialization. The monitor annotation is unfolded into the condition variable \texttt{mon_cond} and its associated lock \texttt{mon_lock}. \texttt{buffer_els} is a bounded integer in the interval \([0,1]\), and is initially set to 1. One \texttt{Producer} and two \texttt{Consumer} threads are spawned with \texttt{start}. Note that some facts, such as thread type and initial values were not annotated; they are inferred automatically by STaVe.

Figure 5.1: STaVe workflow to prove the correct usage of CVs
5.1. OVERVIEW OF THE APPROACH

class Producer extends Thread {
    Buffer buffer;
    Producer(Buffer b){buffer=b;}
    public void run() {
        while (buffer.full())
            buffer.wait();
        buffer.add();
        buffer.notifyAll();
    }
}

class Consumer extends Thread {
    Buffer buffer;
    Consumer(Buffer b){buffer=b;}
    public void run() {
        while (buffer.empty())
            buffer.wait();
        buffer.remove();
        buffer.notifyAll();
    }
}

class Buffer {
    int els, cap;
    void remove(){if (els>0)els--;}
    void add(){if (els<cap)els++;}
    boolean full(){return els==cap;}
    boolean empty(){return els==0;}
}

static void main(String[] s) {
    Buffer b = new Buffer();
    b.els = 1;
    b.cap = 1;
    Consumer c1 = new Consumer(b);
    Consumer c2 = new Consumer(b);
    Producer p = new Producer(b);
    c1.start();
    c2.start();
}

Figure 5.2: Annotated Java program

Figure 5.4 shows the page hierarchy (5.4a) of the CPN extracted from the SyncTask program in Figure 5.3 and samples three of its subpages at the initial configuration. The CPN is composed of several subpages, and the full hierarchical visualization becomes cumbersome. Thus, we also present the complete non-hierarchical version in Appendix D to give an intuitive notion of the program model.

The SyncTask_0 subpage is the top page in the hierarchy. It contains the composition of threads, and the places that represent global variables. The place Start contains two tokens of colour Consumer, and one of colour Producer, representing the spawned threads. The place End collects the tokens representing the terminated threads, and is initially empty. The NotifyAll_Producer_0 subpage presents the CPN component for the notifyAll construct. Among others, it contains the fusion place mon_cond (also present in SyncTask_0), which represents the condi-
CHAPTER 5. VERIFYING SYNCHRONIZATION WITH CONDITION VARIABLES

Thread Producer {
  synchronized(mon_lock){
    while(buf_els==max(buf_els))
      wait(mon_cond);
    if(buf_els<max(buf_els))
      buf_els=(buf_els+1);
    skip;
    notifyAll(mon_cond);
  }
}

Thread Consumer {
  synchronized(mon_lock){
    while((buf_els==0))
      wait(mon_cond);
    if((buf_els>0))
      buf_els=(buf_els-1);
    else
      skip;
    notifyAll(mon_cond);
  }
}

main{
  Lock mon_lock();
  Cond mon_cond(
    mon_lock);
  Int buf_els(0,1,1);
  start(1,Producer);
  start(2,Consumer);
}

Figure 5.3: SyncTask program extracted from annotated Java

The While_Producer_0 subpage presents the component for while. Among others, it contains the buffer_els fusion place (also present in SyncTask_0), representing the global variable buffer_els. It has colour INTO_1, denoting the variable bounds, and contains one token, representing that the variable is initially full.

The analysis of the net (which we explain in Section 5.6) shows that there are no cycles in its reachability graph, and it has a single dead configuration, with the marking of End being three thread tokens. Thus, the program terminates for the given initial values.

5.2 SyncTask

We present the precise definition of SyncTask, an intermediate language that has been designed to specify synchronization schemes, and at the same time to enable underlying verification mechanisms to establish properties about the synchronization. SyncTask abstracts from most features of full-fledged programming languages. For instance, it does not have objects, procedures, exceptions, etc. However, it features the relevant aspects of thread synchronization. We now describe the language syntax, types, and semantics.

5.2.1 Syntax and Types

The SyncTask syntax is presented in Figure 5.5. A program has two main parts: ThreadType*, which declares the different types of parallel execution flows, and Main, which contains the variable declarations and initializations, and defines how the threads are composed, i.e., the static declaration of how many threads of each type are spawned.

Each ThreadType consists of one or more adjacent SyncBlocks, which are mutually exclusive code blocks, guarded by a lock. A code block is defined as a sequence of statements, which may even be another SyncBlock. Notice that this allows nested
5.2. SYNCTASK

SyncBlocks, thus enabling the definition of complex synchronization schemes with more than one lock.

There are four primitive types: reentrant locks (Lock), condition variables (Cond), booleans (Bool), and bounded integers (Int). Expressions are evaluated as in Java. The boolean and integer operators are the standard ones, while max and min return a variable's bounds. Operations between integers with different bounds (overloading) are allowed. However, an out-of-bounds assignment leads the program to an error configuration.
CHAPTER 5. VERIFYING SYNCHRONIZATION WITH CONDITION VARIABLES

SyncTask ::= ThreadType* Main
ThreadType ::= Thread ThreadName { SyncBlock* }
SyncBlock ::= synchronized (VarName) { Block }
Main ::= main { VarDecl* StartThread* }
StartThread ::= start(Const, ThreadName);
VarDecl ::= VarType VarName (Expr*);
VarType ::= Int | Bool | Lock | Cond
Expr ::= Const | VarName | Expr ⊕ Expr | max(VarName) | min(VarName)
Assign ::= VarName = Expr ;
Block ::= { Stmt* }
Stmt ::= Block | SyncBlock | Assign | skip; | if Expr Stmt else Stmt | while Expr Stmt | notifyAll(VarName);
| notify(VarName); | wait(VarName);

Figure 5.5: SyncTask syntax

Condition variables are manipulated by the unary operators wait, notify, and notifyAll. Currently, the language provides only two control flow constructs: while and if-else. These suffice for the illustration of our technique, while the addition of other constructs is straightforward.

The Main block contains the global variable declarations with initializations (VarDecl*), and the thread composition (StartThread*). A variable is defined by its type and name, followed by the initialization, which is enclosed in parentheses. The number of arguments varies per type: Lock takes no arguments; Cond is initialized with a lock variable; Bool takes either a true or a false literal; Int takes three integer literals as arguments: the lower and upper bounds, and the initial value, which must be in the given range. Finally, start takes a positive number and a thread type, signifying the number of threads of that type it spawns.

5.2.2 Structural Operational Semantics

We now describe the structural operational semantics of SyncTask. It is designed to provide an intermediate means to establish formal results between high-level
programming languages and underlying verification mechanisms.

The semantic domains are defined as follows. Booleans are represented as usual. Integer variables are triples $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, where the first two elements are the lower and upper bound, and the third is the current value. A lock $o$ is a pair $(\text{Thread}_\text{id} \cup \{\bot\}) \times \mathbb{N}$ of the id of the thread holding the lock (or $\bot$, if none), and a counter of how many times it was acquired. A condition variable $d$ simply stores its respective lock, which is retrieved with the auxiliary function $\text{lock}(d)$.

SyncTask contains only global variables and all memory operations are synchronized. Thus, we assume the memory to be sequentially consistent [55]. Let $\mu$ represent a program’s memory. We write $\mu[l \mapsto v]$ to denote the update of a variable $l$ with a value $v$, and $\mu(l)$ denotes the value of $l$.

A thread state is either running ($R$) if the thread is executing, waiting ($W$) if it has put itself to sleep on a CV, or notified ($N$) if another thread has woken up the sleeping thread. The state of threads in $W$ or $N$ also store the CV it is/was waiting at, and the number of locks it must reacquire to proceed with the execution. The auxiliary function $\text{waitset}(d)$ returns the ids of all threads waiting on a CV $d$.

We represent a thread as $(\theta, t, X)$, where $\theta$ denotes its id, $t$ denotes the executing code, and $X$ denotes its state. We write $T = (\theta, t, X) | (\theta', t', X') | ...$ for a parallel thread composition. Also, $T[(\theta, t, X)]$ denotes a thread composition, assuming that $\theta$ is not defined in $T$. A program configuration is a pair $(T, \mu)$ of the threads’ composition and its memory. A thread terminates if the program reaches a configuration where its code $t$ is empty ($\epsilon$); a program terminates if all threads in its parallel composition terminate.

The initial configuration is defined by the declarations in Main. As expected, the variable initializations set the initial value of $\mu$. For example, $\text{Int \ i}(lb, ub, v)$ defines a new variable such that $\mu(i) = (lb, ub, v), lb \leq v \leq ub$. The thread composition is defined by the start declarations. E.g., $\text{start}(2, t);$ adds two threads of type $t$ to the thread composition: $(\theta, t, R) | (\theta', t, R)$.

Figure 5.6 presents the operational rules. For readability, we just present the rules for the synchronization statements, as the rules for the remaining statements are standard (see [15, § 3.4-8]). Side conditions are depicted with superscripts, and their definition are presented at the bottom of the figure.

In rule [s1], a thread acquires a lock, if available, i.e., if it is not assigned to any other thread and the counter is zero. The rules [s2] and [s3] represent lock reentrance, and, respectively, increase and decrease the lock counter upon entering and exiting a synchronized block. Rule [s4] applies to the computation of statements inside synchronized blocks. In rule [s5], a thread finishes the execution of a synchronized block, and relinquishes the lock.

In the [wt] rule, a thread changes its state to $W$, stores the counter of the CV’s lock, and releases it. The rules [nf1] and [na1] apply when a thread notifies a CV with an empty wait set; the behaviour is the same as for the skip statement. By rule [nf2], a thread notifies a CV, and one thread in its wait set is selected non-deterministically, and its state is changed to $N$. Rule [na2] is similar, but all threads in the wait set are awaken. By the rule [rd], a thread reacquires all the locks it had
CHAPTER 5. VERIFYING SYNCHRONIZATION WITH CONDITION VARIABLES

[s1] \[ T[\theta, \text{synchronized}(o) b, R], \mu \rightarrow T[\theta, \text{synchronized}(o) b, R], \mu[o \mapsto (\theta, 1)] \]

[s2] \[ T[\theta, \text{synchronized}(o) b, R], \mu \rightarrow T[\theta, \text{synchronized}(o) b, R], \mu[o \mapsto (\theta, n + 1)] \]

[s3] \[ T[\theta, b_1, R, \mu] \rightarrow T[\theta, b_2, X, \mu'] \]

[s4] \[ T[\theta, \text{synchronized}(o) b_1, R), \mu \rightarrow T[\theta, \text{synchronized}(o) b_2, X, \mu'] \]

[s5] \[ T[\theta, \text{synchronized}(o) b, R), \mu \rightarrow T[\theta, \text{synchronized}(o) b, R), \mu[o \mapsto (\bot, 0)] \]

[wt] \[ T[\theta, \text{wait}(d);, R), \mu \rightarrow T[\theta, \text{wait}(W, d, n)), \mu[\text{lock}(d) \mapsto (\bot, 0)] \]

[nf1]\[ T[\theta, \text{notify}(d);, R), \mu \rightarrow T[\theta, \text{notify}(W, d, n)), \mu[\text{lock}(d) \mapsto (\bot, 0)] \]

[nf2]\[ T[\theta, \text{notifyAll}(d);, R), \mu \rightarrow T[\theta, \text{notifyAll}(W, d, n)), \mu[\text{lock}(d) \mapsto (\bot, 0)] \]

[na1]\[ T[\theta, \text{notifyAll}(d);, R), \mu \rightarrow T[\theta, \text{notifyAll}(W, d, n)), \mu[\text{lock}(d) \mapsto (\bot, 0)] \]

[na2]\[ T[\theta, \text{notifyAll}(d);, R), \mu \rightarrow T[\theta, \text{notifyAll}(W, d, n)), \mu[\text{lock}(d) \mapsto (\bot, 0)] \]

[rd] \[ T[\theta, t, (N, d, n)), \mu \rightarrow T[\theta, t, R), \mu[\text{lock}(d) \mapsto (\theta, n)] \]

\[^a\mu(o) = (\bot, 0) \quad ^b\mu(o) = (\theta, n) \land n > 0 \quad ^c\mu(o) = (\theta, n) \land n > 1 \quad ^d\mu(o) = (\theta, 1) \]
\[^e\text{waitset}(d) = \emptyset \quad ^f\theta' \in \text{waitset}(d) \quad ^g\mu(\text{lock}(d)) = (\bot, 0) \]

Figure 5.6: SyncTask operational rules for synchronization

relinquished, changes the state to \( R \), and resumes the execution after the control point where it invoked \text{wait}.

5.3 From Java To SyncTask

We now present the annotation language for specifying the bounded synchronization behaviour of Java programs. It relies on the knowledge about the expected synchronization, and the programmer provides hints for \text{STaVe} to automatically map the synchronization to a SyncTask program.
The annotations are provided in a tree structure, which follows the Java abstract syntax tree (AST). An annotation binds to a specific set of AST nodes. That is, the declaration starts in a comment block immediately above the node declaration, with additional annotations inside the node’s body. Annotations share common keywords (though with a different semantics), and overlap in the node types they may bind to. The ambiguity is resolved by the first keyword (called a switch) found in the comment block. Comments that do not start with a keyword are ignored.

Figure 5.7 presents the annotation language. The text inside square brackets is an optional argument, and the text inside parentheses tells which Java AST node the annotation binds to. The top-level annotations are divided into three categories: resource, synchronization and initialization.

A resource is a data type that is manipulated in the synchronization. It abstracts the state of a data structure to a bounded integer, and defines how the methods operate on it. For example, the annotation abstracts a linked list or a buffer (as in Figure 5.2) by its size. Resources bind to classes only, and the switch @resource

```
Java Program

@resource (classes)
  ___@object [Id -> Id]
  ___@value [Id -> Id]
  ___@capcity [Id -> Id]
  ___@defaulvalue Int
  ___@defaultcap Int
  ___@predicate (MethodDef)
    ___@inline [@maps Id->@{ Code }@]
    ___@code -> @{ Code }@
  ___@operation (MethodDef)
    ___@inline [@maps Id->@{ Code }@]
    ___@code -> @{ Code }@

@syncblock [Id] (synchronized blocks and methods)
  ___@threadtype Id -> Id
  ___@resource Id : ResourceId
  ___@lock Id -> Id
  ___@condvar Id -> Id
  ___@monitor Id -> Id

@synctask [Id] (methods)
  ___@resource Id -> Id
  ___@lock Id -> Id
  ___@condvar Id -> Id
  ___@monitor Id -> Id
  ___@thread [Int : Id]
```

Figure 5.7: Annotation language from Java programs
starts the declaration. \texttt{@value} and \texttt{@capacity} define, respectively, which class member, or \textit{ghost variable}, stores the abstract state, and its maximum value. The keyword \texttt{@operation} binds to method declarations, and expresses that the method potentially alters the resource state. For example, that is the case for the methods \texttt{add} and \texttt{remove} in Figure 5.2. Similarly, \texttt{@predicate} binds to methods, defines that the method returns a predicate about the state, and is exemplified with methods \texttt{empty} and \texttt{full}.

There are two ways to synthesize the annotated method’s behaviour. \texttt{@code} tells \texttt{STaVe} not to process the method, but instead to associate it to the code enclosed between \{ and \}. \texttt{@inline} tells \texttt{STaVe} to try to infer the method declaration, with the potential aid of \texttt{@maps}, which syntactically replaces a Java node (e.g., a method invocation) with a SyncTask code snippet. The above-mentioned methods from Figure 5.2 exemplify the annotation, with \texttt{STaVe} automatically inlining them in the SyncTask program in Figure 5.3.

The \textit{synchronization} annotation defines the observation scope. It binds to \texttt{synchronized} blocks and methods, and the switch \texttt{@syncblock} starts the declaration. Nested synchronization blocks and methods are not annotated; all its information is defined in the top-level annotation. The keywords \texttt{@lock} and \texttt{@condvar} define which mutex and condition object to observe. \texttt{@monitor} has the combined effect of both keywords for an object’s monitor, i.e., a pair of a lock and a CV. Here, \texttt{@resource} maps a local variable to the alias of the global object being observed.

\textit{Initialization} annotations define the global pre-condition for the elements involved in the synchronization, i.e., they define the lock, condition variable and resource declarations with initial value, and the global thread composition. They bind to methods, and the switch \texttt{@synctask} starts their declaration. Here, \texttt{@resource}, \texttt{@lock}, \texttt{@condvar} and \texttt{@monitor} define the objects being observed, and assign global aliases to them. Finally, \texttt{@thread} defines that the following object corresponds to a spawned thread that synchronizes within the observed synchronization objects.

### 5.4 From SyncTask to CPN

Next, we present the modelling of SyncTask constructs as CPN components following the definitions from Section 2.7 and describe how the net is assembled. Our extraction of hierarchical CPNs from SyncTask programs is a variant of the one described in [85].

In our modelling, the colour set \texttt{THREAD} associates a colour to each \texttt{Thread} type declaration, and an individual thread is represented by a token with a colour from the set. Some components are parametrized by \texttt{THREAD}, meaning that they declare transitions, arcs, or places for each thread type. For illustration purposes, we present the parametrized components in an example scenario with three thread types: blue (\texttt{B}), red (\texttt{R}), and yellow (\texttt{Y}). The production rules in Figure 5.5 are mapped into hierarchical CPNs components, where STs represent the non-terminals on the right-hand side.
5.4. FROM SYNCTASK TO CPN

Figure 5.8a shows the top-level component for SyncTask programs. It has the in-socket Start, which contains all threads tokens in the initial configuration, connected by arcs (one per colour) to the STs denoting the thread types, and the out-socket End, which collects the terminated thread tokens. It may also contain the fusion places that represent global variables, as buffer els in Figure 5.4a. Each declaration in ThreadType* (Figure 5.8b) generates a distinct component representing a thread’s control flow. The ST named s1 instantiates a SyncBlock* declaration, with the in-port assigned to Start, and the out-port assigned to End. The sequential composition of consecutive SyncBlocks, and of statements inside a Block declaration is presented in Figure 5.8c.

Figure 5.8: Modelling of SyncTask

Figure 5.9 shows the components for the control flow statements. An if-else statement is modelled by an in-port connected to two transitions, each denoting the evaluation of the control expression to true or false, followed by an in-socket to the respective ST denoting the respective ‘then’ or ‘else’ block, and arcs connecting to out-port. The while statement is modelled by an in-port, denoting a control point immediately before the expression evaluation, connected to two transitions: one is enabled if the expression is false, followed by an out-port denoting the control point after the loop; the other one is enabled if the expression is true, followed by an in-socket to the ST denoting the loop body, and an arc to the in-port, denoting expression re-evaluation.
CHAPTER 5. VERIFYING SYNCHRONIZATION WITH CONDITION VARIABLES

Figure 5.10 shows the components for the synchronization primitives. A SyncBlock is modelled with a single in-port, a transition denoting lock acquisition, an in-socket denoting the critical section entrance, a ST denoting the body declaration, a transition denoting the lock release, and an out-socket denoting the exit from the critical section. A wait is modelled as a transition that produces two tokens: one into the place modelling the CV, and one into the place modelling the lock, representing its release; a place for the woken up threads, and a transition to reacquire the lock, and an out-port, denoting the control point where threads resume the execution. A notify is modelled by a transition that is enabled if the CV is empty, plus one transition and one out-port per colour, modelling the non-deterministic choice of which thread to wake, and the routing of tokens to the place to reacquire the lock. A notifyAll is similar, but the transition that checks if the CV is empty is enabled after all thread tokens have been woken up.

CPN Tools is integrated with an ML-based engine [44] for expression evaluation, analogously to model checkers and SMT-solvers. Thus, in the current modelling, boolean and integer expressions are conveniently translated to ML expressions, and assigned to transitions (for branching) and arcs (for assignments).

The global variable declaration \texttt{VarDecl\*} generates a place containing a single token for each \texttt{Lock} object. An empty place denotes that some thread holds the lock. We define the colour set \texttt{CPOINT} with colours representing the control points with a wait statement. A \texttt{Condition} variable generates an empty place denoting the waiting set, with colour set \texttt{CONDITION}. Here, colours are pairs of \texttt{THREAD} and \texttt{CPOINT}. Both data are necessary to route correctly woken up threads to the correct place where they resume execution. A \texttt{Bool} variable generates a place with

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{control_flow_structures.png}
\caption{Modelling of control flow structures}
\end{figure}
5.4. FROM SYNCTASK TO CPN

Figure 5.10: Locking acquisition/release, and signaling with condition variables

colour set BOOL. An Int variable generates a place and a new colour set of bounded integers, with colours being numbers in the variable’s range.

The initialization in main does not produce places or transitions. It simply declares the initial set of tokens for the places representing variables, and the number and colours of thread tokens. As seen in the Start place in Figure 5.4, a marking is depicted textually on top of the place. It declares pairs of tokens and colours, with ++ being the separator.
5.5 Correctness

In this section we outline the theoretical results that, combined, entail the correctness of our framework. The complete proofs, though straightforward in their essence, are long and the details are tedious. Therefore, here we simply present the intuitive notion behind them, and refer to \[20\] for the full account.

We are interested in proving that all threads synchronizing under a given set of condition variables will eventually finish the synchronization, and proceed with the execution. Thus, we condition the correct synchronization to all threads invariably reaching their synchronization blocks. Existing techniques for programs without condition variables (e.g., \[67\]) can establish the liveness property that threads will reach a given control point, and such proofs are out of the scope of this work. We define the correct synchronization of a program as follows.

**Definition 16** (Correct synchronization). Given a Java program with finite sets of condition variables and threads, its synchronization is correct if, whenever every thread eventually reaches the entry point of its synchronization block, then every thread will eventually reach the first control point after the block.

It is important to stress that the definition above applies to a one-time synchronization of the Java program. However, if it can be proven that the initial conditions are the same every time the synchronization scheme is spawned, then the scheme is correct for an arbitrary number of invocations. This may be proven by showing that a Java program always resets the variables observed in the synchronization before re-spawning the threads.

Additionally, we relativize the verification of correct synchronization on the correctness of the user-provided annotations. Thus, also the annotations must be correct, and reflect the actual synchronization behaviour. Here, we informally define the notion of correct annotation as being the one that defines an abstraction to a SyncTask program s.t. there exits a weak bisimulation to the Java synchronization. There exist techniques to verify the facts provided in the annotations, such as the bounds of a variable, number of threads, and mappings of data structures to predicates. Thus, although it is out of the scope of the present work, we speculate that STAVe may be integrated with static analysis tools to prove the correctness of the annotations, and thus, to entail the full proof of correct synchronization.

We now define the relation between a correctly annotated synchronization in a Java program and the termination of its respective SyncTask program.

**Proposition 2** (SyncTask termination). Given a correctly annotated Java program, its synchronization is correct iff the corresponding SyncTask program terminates.

The annotations delimit an abstraction of the synchronization. I.e., they define an abstraction function from program variables to predicates (e.g., the size of a list) and bounded variables. The SyncTask operational semantics presented in Section \[5.2\] has been defined to mimic closely the semantics defined in \[15\] for...
5.6. THE STaVe TOOL

Java. Therefore, if the annotations are correct, one can prove by means of weak bisimulation that any change in an abstract variable that is observed is also captured by the corresponding variable in SyncTask, and vice versa, similarly to a monitor.

Now we enunciate the result that connects the termination of a SyncTask program to its corresponding Coloured Petri Net invariably reaching a special set of dead configurations.

**Proposition 3** (CPN termination). A SyncTask program terminates iff its corresponding CPN always reaches dead configurations where the \textbf{End} place has the same marking as the \textbf{Start} place in the initial configuration.

A CPN declares a place for each SyncTask variable. Moreover, there is a clear correspondence between the operational semantics of a SyncTask construct and its respective CPN component. One can show by means of weak bisimulation that every configuration of a SyncTask program is matched by a unique set of consecutive CPN configurations. This implies that if the \textbf{End} place in all dead configurations has the same tokens as the \textbf{Start} place in the initial configuration, then for every possible scheduling, every thread in the SyncTask program terminates its execution. Note that the non-deterministic thread scheduler is simulated by the non-deterministic firing of transitions.

The CPN termination can be verified algorithmically with the computation of the reachability graph from the generated CPN, and the checks that (1) there are no cycles, (2) that the only reachable dead configurations are the ones where the marking in the \textbf{End} place is the same as the marking in the \textbf{Start} place in the initial configuration, and (3) that there is at least one dead configuration (i.e., not vacuously true).

5.6 The STaVe tool

STaVe implements the parsing of annotated Java (and also SyncTask) source programs for generating ASTs of SyncTask programs, and the extraction of hierarchical CPNs from the ASTs. The tool has been written in Java, and is available at [24].

STaVe processes the annotations in an intricate scheme. It takes the annotated Java program as input, and uses the JavaParser library to generate the AST. Then it converts the JavaParser’s AST into the one of the OpenJDK compiler, to take advantage of its symbol table querying, type checking and code optimization. We have adopted JavaParser for the parsing because it associates the comments per-AST node, while OpenJDK’s parser discards annotations of a finer granularity than methods. This allows the annotation of \textit{synchronized} blocks. Next, STaVe traverses the Java AST three times, processing resource, initialization, and synchronization, in this order, to finally output the SyncTask AST. Then, it extracts the CPN from the AST following the mapping between SyncTask syntax and CPN components described in Section 5.4.
CHAPTER 5. VERIFYING SYNCHRONIZATION WITH CONDITION VARIABLES

1: CalculateOccGraph ();
2: CalculateSccGraph ();
3: NoOfNodes ();
4: fun terminal node =
   Mark.SyncTask_0 'End 1 node =
   Mark.SyncTask_0 'Start 1 InitNode;
5: not (List.null(ListDeadMarkings ()))
   andalso List.all (terminal) (ListDeadMarkings ())
   andalso (NoOfNodes () = SccNoOfNodes ());

Figure 5.11: ML code for generating and querying a CPN state space

Two parts of STaVe turned out to be useful for others projects, and became spin-offs. The first is JavaParser2JCTree [22], a library that translates JavaParser ASTs to OpenJDK ASTs. The second is libcpntools [23], a library that generates hierarchical CPNs in the CPN Tools’s XML-based file format.

We now describe the experimental evaluation of our framework. This includes the process of annotating Java programs, extraction of the corresponding CPNs, and the analysis of the nets using CPN Tools. To evaluate the scalability of STaVe w.r.t. the size of the part of program that does not affect the synchronization, we take a test case that is a rather large program, but which has simple synchronization behaviour. Next, to evaluate the scalability of our tool w.r.t. the number of synchronizing threads, we take the example program from Section 5.1, and instantiate it with varying parameters.

In the first test case we annotate PIPE [27] (version 4.3.2), a CPN analysis tool written in Java. It contains a single synchronization scheme using CVs, where a thread that sends logs to a client via a socket waits for a server thread to establish the connection, and then notifies. This test case illustrates that synchronization involving CVs is typically simple and bounded. Manually annotating the program took just a few minutes, once the synchronization scheme was understood. The CPN extraction time was also negligible, and the verification process took just a few milliseconds to establish the correctness.

In the second test case, we performed experiments on the example presented in Section 5.1, with a varying number of threads, buffer capacity, and initial value. The experiments were performed in a Windows 7 virtual machine with 8GB of RAM and 2 processors, running on top a Linux machine with 16G and a quad-core Intel i5 CPU of 1.30GHz.

CPN Tools provides an ML API [43] for generating and querying the state space. We collect our statistics by executing the code in Figure 5.11. Line 1 generates the state graph, from which the command on Line 2 computes the strongly connected components. Line 3 queries the reachable configurations. Line 4 defines a function which receives a configuration as parameter, and checks if the End place (in the
SyncTask_0 subpage) has the same marking as the Start place in the initial configuration (InitNode). Line 5 checks the three termination conditions, namely: whether there is at least one dead configuration; whether all dead configurations respect the condition defined on Line 4, and whether the number of strongly connected components is equal to the size of the state graph, implying the absence of cycles.

Table 5.1 presents the practical evaluation for a number of initial configurations. We observe an expected correlation between the number of tokens representing threads, the size of the state space, and the verification time. However, we have observed an unexpected influence of varying buffer capacities and initial states. We conjecture that the initial configurations that model high contention, i.e., many threads waiting on CVs, induce a larger state space. The experiments also show how the termination depends on the thread composition and initial state, meaning that a single change in any parameter may affect the verification result.

5.7 Related Work

Leino et al. [56] propose a compositional technique to verify the absence of deadlocks in concurrent systems with both locks and channels. They use deductive reasoning to define which locks a thread may acquire, or to impose an obligation for a thread to send a message. The authors acknowledge that their quantitative approach to channels does not apply for CVs because messages passed through a channel are received synchronously. But a notification on a condition variable is either received, or it is lost.

Popeea and Rybalchenko [67] present a compositional technique to prove termination of multi-threaded programs, which combines predicate abstraction and refinement with rely-guarantee reasoning. The technique is only defined for programs that synchronize with locks, and it cannot be easily lifted to support CVs. The reason is that the thread termination criterion is the absence of infinite computations. However, a finite computation where a waiting thread is never notified is incorrectly characterized as terminating.

Wang and Hoang [84] propose a technique that permutes actions of execution traces to verify the absence of synchronization bugs. Their program model considers locks and condition variables. However, they cannot verify the same property as ours, since their method does not permute matching pairs of wait-notify. For instance, it will not reorder a trace where, first, a thread waits, then another thread notifies. Thus, their method cannot detect the case where the notifying thread is scheduled first, and the waiting thread suspends the execution indefinitely. Also, the presented test cases only verify bugs caused exclusively by the incorrect usage of locks.

Kaiser and Pradat-Peyre [38] propose the modelling of Java monitors in Ada, and the extraction of CPNs from Ada programs. However, they do not precisely describe how the CPNs are verified, nor provide a correctness argument about their
Table 5.1: Statistics for Producer/Consumer

<table>
<thead>
<tr>
<th>Initial Configuration</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Buffer</td>
</tr>
<tr>
<td>Producer</td>
<td>Consumer</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>26</td>
<td>24</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>18</td>
</tr>
</tbody>
</table>

Kavi et al. [49] present PN components for the synchronization primitives in the Pthread library for C/C++, including condition variables. However, their modelling of CV just allows the synchronization between two threads, and no argument is presented on how to use it with more threads.
5.7. RELATED WORK

Westergaard [85] presents a CPN extraction technique for a simple parallel toy language containing locks only. Our work borrows much from this work w.r.t. the modelling and analysis of Coloured Petri Nets. However, we analyze full-fledged programming languages, and address the complications of analyzing programs with condition variables.

Aalst et al. [83] present strategies for modelling complex parallel applications as CPNs. We benefit from many ideas from this work, especially the modelling of hierarchical CPNs. However, their formalism is over-complicated for our needs. Thus, we simplify it to produce manageable CPNs.
Chapter 6

Conclusion

In this thesis we have presented techniques that involve the automatic generation of abstract models for the formal verification of programs in fully-fledged languages. Together, they cover a wide range of program properties, program models, verification scenarios, sequential and parallel programs, source and executable code. All the techniques address both theoretical and practical aspects involving the extraction of program models, and have been implemented in tools that have been validated over real-world test cases.

The first technique presents an efficient and precise control flow graph extraction algorithm for sequential Java bytecode programs that considers normal as well as exceptional control flow.

The algorithm is defined by means of a translation to the intermediate, stackless Java bytecode representation BIR, supported by the Sawja tool. The CFGs are then extracted from the BIR representation through a combination of intra-procedural and inter-procedural analyses. The constructed CFGs are precise, relative to the chosen algorithm for virtual method call resolution and to static analyses that abstract from all data but exceptions.

The main contribution of the technique is a formalization and a soundness proof, providing a formal argument for why the algorithm is correct, in the sense that it extracts control flow graphs whose behaviour (in terms of sequential program executions) soundly over-approximates the behaviour of the original program. To the best of our knowledge, this is the first CFG extraction algorithm that has been proven correct. This makes the extracted CFGs especially useful for a technique for the compositional verification of temporal control flow safety properties, but also for other types of program analyses.

The correctness proof of the algorithm is simplified by reusing two previous theoretical results. The first establishes that structural simulation entails behavioural simulation. The second is an idealized CFG extraction algorithm from JBC, which has been proven to simulate structurally the JBC, and, following from the first result, also simulates the JVM behaviourally. Therefore, we establish structural
(rather than behavioural) simulation between the CFGs extracted by the idealized algorithm and the CFGs extracted by our algorithm. Still, the proof is not trivial, and requires the introduction of the notion of bytecode segment, in order to match bytecode instructions to BIR instructions. Structural simulation is then shown for the CFGs extracted by the two algorithms for each segment of the original program, by case analysis on the type of the last instruction in the segment. The proof is presented in pencil-and-paper style, but paves the ground for a mechanized proof using a standard theorem prover.

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

The second technique presents a framework to extract control flow graphs from the available components of incomplete Java bytecode programs.

We have introduced a formal representation of incomplete JBC programs where unavailable components are modelled by user-provided interfaces. Next, we have extended the previously defined algorithm to extract CFGs from the available software components, which now uses the provided interfaces to resolve dependencies involving unavailable components. 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.

We formally define a refinement relation over Java bytecode programs through a set of constraints to instantiate the unavailable components, and show that the relation is property preserving. That is, if the relation holds between an incomplete program, and a complete program that implements it, then the control flow graphs extracted from the available components in the incomplete program structurally simulate the control flow graphs extracted from the same components in the complete program. Therefore, safety properties established over the extracted CFGs hold for complete program that implement the original incomplete system. To the best of our knowledge, our framework is the first to produce control flow graphs for the available components of an incomplete program that are property preserving.

ConFlEx has been extended to support the framework. It features the modular extraction algorithm from (incomplete) BIR programs, and the caching of previous analyses, thus enabling incremental CFG extraction. Also, Sawja has been instrumented to fit our needs. We have implemented a type analysis for explicit exceptions, a sound virtual method call resolution algorithm for modular set-ups, the representation of incomplete programs, and the check of refinement.

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 CFGs for incomplete programs, especially when only few components are missing.

The third technique presents a formal framework for the automatic verification of the correct usage of condition variables in concurrent Java programs.

Here, correctness means “if all threads synchronizing under the same set of condition variables eventually enter their synchronization block, then all threads will exit their blocks”. The framework uses Coloured Petri Nets as program model, and has been implemented as the STaVe tool.

Our technique is based on SyncTask, an intermediate language that captures bounded synchronization behaviour, and thus for the programs in this language termination becomes decidable. The tool extracts Coloured Petri Nets from SyncTask programs, and reduces termination of the latter to a reachability problem on the nets; a check that is delegated to CPN Tools. In case the verification fails, it provides a sequence of CPN configurations that explains the error to the programmer. Test cases show that the CPN analysis suffers from the combinatorial explosion of the state space, which limits the size of synchronization schemes that can be verified. Nevertheless, it has scaled up to a scenario with fifty threads, and is thus able to verify intricate synchronization schemes.

Our tool provides an annotation language for programmers to stipulate the expected synchronization of Java programs. The annotations are minimal, in the sense that the user only has to provide the information that cannot be automatically inferred by STaVe. The tool extracts SyncTask programs from annotated Java programs, and reduces their correct synchronization to the termination of the SyncTask program. A test case on a large Java program illustrates that the typical synchronization behaviour is bounded, and shows STaVe’s utility for verifying real-world programs.
Bibliography


Appendix A

Correctness of $G_{bir} \circ BC2BIR$

The $BC2BIR$ transformation may collapse several bytecode instructions into a single BIR instruction. We divide the JBC instructions into two sets: the producer instructions, i.e., those that produce at least one BIR instruction in function $BC2BIR_{instr}$, and the auxiliary ones, i.e., those that produce none. This division can be deduced from Figure 2.13 (on page 29). For example, store and invokevirtual are producer instructions, while add and push are auxiliary.

We partition the bytecode instruction array into bytecode segments. These are sub sequences delimited by producer instructions. Thus, each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Equivalently, we may say that a segment is defined by an optional basic block and a subsequent producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java Bytecode Verifier (see Section 2.1). All methods in such programs must have goto, return, or athrow as the last (reachable) instruction, which are producer instructions. Therefore, there can not be contiguous (reachable) instructions that are not delimited by a producer.

Each bytecode segment is transformed into a set of contiguous instructions by $BC2BIR$. We call this set a BIR segment, which is a partition of the BIR instruction array. There exists a one-to-one mapping between bytecode segments and the BIR segments, which is also order-preserving. Thus, we can associate each instruction, either in the JBC or BIR arrays, to the unique index of its correspondent bytecode segment. Figure 2.14 (on page 31) illustrates the partitioning of instructions into segments. Method odd has eight bytecode (and BIR) segments, as indicated by the distinct shades. Producer instructions are underlined.

In the definition of the direct extraction algorithm in Figure 2.10 (on page 24), one can observe that all auxiliary instructions give rise to an internal transfer edge only. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path-like graph of internal transfer edges of the same length as the number of auxiliary instructions, followed by the edges generated for the producer instruction. Let $p$ be the position for the first auxiliary instruction,
and $q$ the position of the producer instruction. We illustrate the pattern for this path-like graph below.

$$
\circ_m^p \xrightarrow{e} \circ_m^{\text{succ}(p)} \xrightarrow{e} \circ_m^{\text{succ}(\text{succ}(p))} \xrightarrow{e} \ldots \xrightarrow{e} \circ_m^q
$$

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

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

Based on the observations above, our main theorem states that the method graphs extracted using the indirect algorithm weakly simulate (see Definition 9) the method graphs using the direct algorithm. The abstract stacks are omitted in the proof, since examining the instructions is sufficient to produce the edges.

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

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

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

We enunciate Definition 11 again, which defines the binary relation $R$ as the union of two binary relations $R_1$ and $R_2$, and show the relation to be a weak simulation in the standard fashion, following Proposition 1 (page 35): for every pair of nodes in $R$, we first show that the nodes have the same set of atomic propositions, and then match every strong edge that has the first node as source, to a corresponding weak edge that has the second node as source, so that the target nodes are again related by $R$. 


Definition (Relation Between CFGs nodes).

\[ R \overset{\text{def}}{=} R_1 \cup R_2 \]

\[ R_1 \overset{\text{def}}{=} \{(o^p_m, o^p_c) \mid \text{seg}_{\text{JBC}}(m, p) = \text{seg}_{\text{BIR}}(m, pc) \land pc = \text{fst}(\text{seg}_{\text{BIR}}(m, pc))\} \]

\[ R_2 \overset{\text{def}}{=} \{(o^{XR}_m, o^{pc XR}_m) \mid \text{seg}_{\text{JBC}}(m, p) = \text{seg}_{\text{BIR}}(m, pc) \land pc \in \text{oap}(\text{seg}_{\text{BIR}}(m, pc), XR)\} \]

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

Let \((o^p_m, *) \in R_1\) and let \((o^{p,r}_m, *) \in R_2\) where \(o^p_m\) and \(o^{p,r}_m\) are control nodes in \(G_{\text{JBC}}(\Gamma[m])\) and \(*\) is a control node in \((G_{\text{BIR}} \circ BC2BIR)(\Gamma[m])\). We consider the two cases separately. Let first \((o^p_m, *) \in R_1\). The proof proceeds by case analysis on the type of the producer instruction of the bytecode segment \(\text{seg}_{\text{JBC}}(m, p)\) giving rise to \(o^p_m\). The cases follow the subsets of JBC instructions presented in Figure 2.9, which share the same extraction rule in the direct algorithm for its instructions. We rely on the notation to indicate that two nodes have the same set of atomic propositions.

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

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

\[ G_{\text{JBC}}^{m,p,\text{store}} = \{o^p_m \xrightarrow{\varepsilon} o^{\text{succ}(p)}_m\} \]

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

Subcase I \(BC2BIR_{\text{instr}}\) produces a single assignment for \texttt{store}. Applying the Assignment rule of \(G_{\text{BIR}}\), we obtain a single edge:

\[ BC2BIR_{\text{instr}}^{m,p,\text{store}} = \{[l_j := e]\} \]

\[ G_{\text{BIR}}^{m,pc,[1:j:=e]} = \{o^p_pc \xrightarrow{\varepsilon} o^{pc+1}_m\} \]
APPENDIX A. CORRECTNESS OF $G_{\text{BIR}} \circ \text{BC2BIR}$

Then $\ast = \circ_m^p$ since $p = \text{fst}(s)$. There is the edge $\circ_m^p \xrightarrow{\circ_{\text{succ}(p)}} \circ_{m+1}^{p+1}$. Moreover, also $(\circ_m^{\text{succ}(p)}, \circ_m^{p+1}) \in R$ since $\text{seg}_{\text{BC}}(m, \text{succ}(p)) = \text{seg}_{\text{BIR}}(m, p+1)$ and $p+1 = \text{fst}(\text{seg}_{\text{BIR}}(m, p+1))$. That is, $p+1$ is the first control address in the next code segment. The case for $\text{nop}$ is analogous to this.

Subcase II $\text{BC2BIR}_{\text{instr}}$ produces two assignments. Applying the Assignment rule from $G_{\text{inst}}$ twice, we have:

$$G_{\text{BIR}}^{m, \text{pc}[t_i:=t_j]} = \{ \circ_m^e \xrightarrow{\circ^{p=1}} \circ_m^{p+1} \}$$

$$G_{\text{BIR}}^{m, \text{pc}[l_i:=l_j]} = \{ \circ_m^e \xrightarrow{\circ^{p+1} \circ^p} \circ_m^{p+2} \}$$

Then $\ast = \circ_m^p$ since $p = \text{fst}(s)$. There is an edge $\circ_m^p \xrightarrow{\circ_{\text{succ}(p)}} \circ_{m+1}^{\text{succ}(p)}$. It is matched by the weak edge $\circ_m^p \xrightarrow{\circ_m^{p+1}} \circ_{m+1}^{p+2}$, which traverses $\circ_m^{p+1}$. Then, also $(\circ_m^{\text{succ}(p)}, \circ_m^{p+2}) \in R$ since $\text{seg}_{\text{BC}}(m, \text{succ}(p)) = \text{seg}_{\text{BIR}}(m, p+2)$ and $p+2 = \text{fst}(\text{seg}_{\text{BIR}}(m, p+2))$. That is, $p+2$ is the first control address in the next code segment.

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

$$G_{\text{BIR}}^{m, p, \text{if}[\text{expr} \ x]} = \{ \circ_m^p \xrightarrow{\circ_{\text{succ}(p)}} \circ_m^q \}$$

The transformation $\text{BC2BIR}_{\text{instr}}$ returns a single instruction, applied to which $G_{\text{inst}}$ produces two edges:

$$G_{\text{BIR}}^{m, p, \text{if}[\text{expr} \ x]} = [\text{if} \ \text{expr} \ x']$$

Then $\ast = \circ_m^p$ since $p = \text{fst}(s)$. The first edge $\circ_m^p \xrightarrow{\circ_{\text{succ}(p)}} \circ_m^{p+1}$ is matched by $\circ_m^p \xrightarrow{\circ_m^{p+1}} \circ_m^{p+2}$. Moreover, $(\circ_m^{\text{succ}(p)}, \circ_m^{p+1}) \in R$ since $\text{seg}_{\text{BC}}(m, \text{succ}(p)) = \text{seg}_{\text{BIR}}(m, p+1)$ and $p+1 = \text{fst}(\text{seg}_{\text{BIR}}(m, p+1))$. The second edge $\circ_m^p \xrightarrow{\circ_{\text{succ}(p)}} \circ_m^q$ is matched by $\circ_m^p \xrightarrow{\circ_m^q}$, and again $(\circ_m^{\text{succ}(p)}, \circ_m^q) \in R$ since $\text{seg}_{\text{BC}}(m, q) = \text{seg}_{\text{BIR}}(m, q')$ and $q' = \text{fst}(\text{seg}_{\text{BIR}}(m, q'))$. That is, both $p+1$ and $q'$ are the first control addresses in their respective code segments.

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

$$G_{\text{BIR}}^{m, p, \text{goto}[\text{expr} \ x]} = \{ \circ_m^p \xrightarrow{\circ_{m}} \circ_m^q \}$$
The transformation $BC2BIR_{mstr}$ returns a single instruction, applied to which $G_{bir}$ produces a single edge:

$$BC2BIR_{mstr}^{\text{goto } q} = \text{[goto } pc']$$

$$G_{bin}^{m,pc,[\text{goto } pc']} = \{ e_m \xrightarrow{\ } o_m^{pc'} \}$$

Then $* = o_m^{pc'}$ since $pc = \text{fst}(s)$. There is the edge $o_m^{pc} \xrightarrow{\ } o_m^{pc'}$, which is matched by the corresponding edge $o_m^{pc} \Rightarrow o_m^{pc'}$. Moreover, $(o_m^{pc}, o_m^{pc'}) \in R$ since $\text{seg}_{BIR}(m, q) = \text{seg}_{BIR}(m, pc')$ and $pc' = \text{fst}(\text{seg}_{BIR}(m, pc'))$. That is, $pc'$ is the first control address on its code segment.

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

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

$$G_{bin}^{m,p,\text{throw } x} = \begin{cases} \{ e_m \xrightarrow{\ } \bullet_m^{p,x} \xrightarrow{\ } o_m^{pc'} \} & \text{there is a handler} \\ \{ e_m \xrightarrow{\ } \bullet_m^{p,x,r} \} & \text{otherwise} \end{cases}$$

The transformation $BC2BIR_{mstr}$ returns a single instruction. Then, similarly to $G_{bin}$, the $G_{bin}$ function produces either one, or two edges:

$$BC2BIR_{mstr}^{\text{asthrow}} = \text{[throw } e]$$

$$G_{bin}^{m,pc,[\text{throw } e]} = \begin{cases} \{ e_m \xrightarrow{\ } \bullet_m^{pc,x} \xrightarrow{\ } o_m^{pc'} \} & \text{there is a handler} \\ \{ e_m \xrightarrow{\ } \bullet_m^{pc,x,r} \} & \text{otherwise} \end{cases}$$

Then $* = o_m^{pc'}$ since $pc = \text{fst}(s)$. In the case where there is an exception handler for $x$ in $p$ and $pc$, the edge $o_m^{pc} \xrightarrow{\ } \bullet_m^{p,x}$ is matched by the corresponding weak edge $o_m^{pc} \Rightarrow \bullet_m^{p,x}$. Then also $(\bullet_m^{p,x}, \bullet_m^{pc,x}) \in R$ since $\text{seg}_{BIR}(p) = \text{seg}_{BIR}(m, pc)$ and $pc \in \text{op}(s, \{ x \})$. That is, $pc$ tags a node where the set $XP = \{ x \}$. Actually, for this case there is only one node since $pc$ is the only control address in the segment. Further, there is the edge $\bullet_m^{p,x} \xrightarrow{\ } o_m^{pc'}$, which is matched by $\bullet_m^{p,x} \Rightarrow o_m^{pc'}$, and again $(o_m^{pc}, o_m^{pc'}) \in R$ since $\text{seg}_{BIR}(m, q) = \text{seg}_{BIR}(m, pc')$ and $pc' = \text{fst}(\text{seg}_{BIR}(m, pc'))$. That is, $pc'$ is the first control address on its code segment.

In the case where there is no exception handler for $x$, the only edge is $o_m^{pc} \xrightarrow{\ } \bullet_m^{p,x,r}$, which is matched by $o_m^{pc} \Rightarrow \bullet_m^{pc,x,r}$. Moreover, $(\bullet_m^{p,x,r}, o_m^{pc,x,r}) \in R$ since $\text{seg}_{BIR}(p) = \text{seg}_{BIR}(m, pc)$ and $pc \in \text{op}(s, \{ x, r \})$. That is, $pc$ tags the node where the set $XP = \{ x, r \}$, which concludes the case.
Case \( i \in \text{XmpInst} \) The instructions in this set follow to the next control point in case they terminate the execution normally, or can raise an exception if some condition was violated. We present this case for the \( \text{div} \) instruction, which can only raise \( x = \text{ArithmeticException} \). The cases for all other instructions in \( \text{XmpInst} \) are analogous.

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

\[
\mathcal{G}_{\text{BIR}}^{m,p,\text{div}} = \begin{cases} \{ \phi_{m} \xrightarrow{\phi} \phi_{mp} \}, \phi_{m} \xrightarrow{\phi} \phi_{mp}, \phi_{mp} \xrightarrow{\phi} \phi_{mp} \} & \text{there is a handler} \\ \{ \phi_{m} \xrightarrow{\phi} \phi_{mp} \}, \phi_{m} \xrightarrow{\phi} \phi_{mp}, \phi_{mp} \xrightarrow{\phi} \phi_{mp} \} & \text{otherwise} \end{cases}
\]

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

\[
\mathcal{G}_{\text{BIR}}^{m,pc,\text{[notzero]}} = \begin{cases} \{ \phi_{m} \xrightarrow{\phi} \phi_{mp} \}, \phi_{m} \xrightarrow{\phi} \phi_{mp}, \phi_{mp} \xrightarrow{\phi} \phi_{mp} \} & \text{there is a handler} \\ \{ \phi_{m} \xrightarrow{\phi} \phi_{mp} \}, \phi_{m} \xrightarrow{\phi} \phi_{mp}, \phi_{mp} \xrightarrow{\phi} \phi_{mp} \} & \text{otherwise} \end{cases}
\]

Then \( x = \phi_{mp} \) since \( pc = \text{fst}(s) \). The edge \( \phi_{mp} \xrightarrow{\phi} \phi_{mp+1} \) is matched by \( \phi_{mp} \xrightarrow{\phi} \phi_{mp+1} \) in both cases. Moreover, also \( (\phi_{mp} \xrightarrow{\phi} \phi_{mp+1}) \in R \) since \( \text{seg}_{\text{IRC}}(m, \text{succ}(p)) = \text{seg}_{\text{BIR}}(m, pc+1) \) and \( pc+1 = \text{fst}(\text{seg}_{\text{BIR}}(m, pc+1)) \). That is, \( pc+1 \) is the first control address in the next segment.

In the case where there is an exception handler for \( x \), there is the edge \( \phi_{mp} \xrightarrow{\phi} \phi_{mp} \), which is matched by \( \phi_{mp} \xrightarrow{\phi} \phi_{mp} \). Thus, also \( (\phi_{mp} \xrightarrow{\phi} \phi_{mp}) \in R \) since \( \text{seg}_{\text{IRC}}(m, p) = \text{seg}_{\text{BIR}}(m, \text{pc}) \) and \( \text{pc} \in \text{oap}(s, \{x\}) \). Similarly to \( \text{ThrInst} \), in this case \( \text{pc} \) is the only control address tagging a node also tagged by \( x \) since it is the only address in the segment. Also, there is the edge \( \phi_{mp} \xrightarrow{\phi} \phi_{mp} \), which is matched by \( \phi_{mp} \xrightarrow{\phi} \phi_{mp} \), and again \( (\phi_{mp} \xrightarrow{\phi} \phi_{mp}) \in R \) since \( \text{seg}_{\text{IRC}}(m, q) = \text{seg}_{\text{BIR}}(m, \text{pc'}) \) and \( \text{pc'} = \text{fst}(\text{seg}_{\text{BIR}}(m, pc')) \).

If there is no exception handler for \( x \), then the direct algorithm produces the edge \( \phi_{mp} \xrightarrow{\phi} \phi_{mp} \), which is matched by \( \phi_{mp} \xrightarrow{\phi} \phi_{mp} \). Then, \( (\phi_{mp} \xrightarrow{\phi} \phi_{mp}) \in R \) since \( \text{seg}_{\text{IRC}}(p) = \text{seg}_{\text{BIR}}(m, \text{pc}) \) and \( \text{pc} \in \text{oap}(s, \{x, r\}) \). That is, \( \text{pc} \) tags the node where the set \( XP = \{x, r\} \), which concludes the case.

Case \( i \in \text{InvInst} \) This is the subset of instructions that execute method invocations: \text{invoke}, \text{invokespecial}, \text{invokestatic}, \text{invokevirtual}, and \text{invokeinterface}.

The first two instructions always have a single receiver for the invocation, while
the last two instructions may have more than one potential receiver, because of the dynamic dispatching.

This is reflected in the function $Rec^Γ_Γ$ in Figure 2.10 which uses a virtual method call algorithm $α$ to list the possible receivers for a method invocation. The indirect algorithm has an equivalent function, $res^α$, which returns the single receiver for a non-virtual method call, or uses the same algorithm $α$ to list the possible receivers of a virtual call. The set of possible receivers for a method invocation is the same for both algorithms. Therefore it suffices to show that, for a (potential) receiver of a method invocation, the subgraph extracted indirectly weakly simulates the subgraph extracted directly.

We present the proof for the single receiver of an $invokespecial$ call. We chose this instruction because it is the only one that may produce two distinct sets of BIR instructions (see Figure 2.13). We thus consider two subcases: (I) when the receiver is an object constructor, and (II), when it is a private method or a method from the super class. For the latter, the BIR instructions are the same as for $invokestatic$, $invokevirtual$, and $invokeinterface$, and the proofs are analogous.

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

\[
\begin{align*}
\mathcal{G}_{m,p,invokespecial} & = \\
& \left\{ \text{there is a handler} \right\} \\
& \left\{ \text{otherwise} \right\} \\
& \left\{ \text{there is a handler} \right\} \\
& \left\{ \text{otherwise} \right\}
\end{align*}
\]

Subcase I The receiver of an $invokespecial$ is an object constructor. We instantiate the signature of the invoked method to $n = C$ to stress this. It returns a sequence of assignments to temporary variables ($[t_1 := l_1; t_2 := l_2; \ldots ]$), denoted by $HSave$; plus the call to [new $C$].

\[
\text{BC2BIR}_{m,invokespecial} = [HSave(pc, as); t_k := \text{new } C]
\]

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

$$G^{m,pc, H\text{Save}(pc, as)} = \{ (m, \phi_m^C, \phi_m'^{C'}) \}$$

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

$$\phi_m^{m, pc', \{t_x = \text{new } C\}} = 
\begin{cases} 
\{ (m, \phi_m^C, \phi_m'^{C'}) \} & \text{there is a handler} \\
\phi_m^C \cup \phi_m'^{C'} \cup \phi_m^{N.P.E., R.P.E., R} \cup \phi_m^{N.P.E., C} & \text{otherwise}
\end{cases}$$

Then $s = \phi_m^C$ since $pc = \text{fst}(s)$. The edge $\phi_m^p \rightarrow \text{suc}(p)$ is matched by $\phi_m^C \rightarrow \phi_m'^{C'}$, which traverses all the nodes extracted from $H\text{Save}$, and also $(\phi_m^C, \phi_m'^{C'}) \in R$ since $\text{seg}_{BC}(m, \text{suc}(p)) = \text{seg}_{BIR}(m, pc' + 1)$ and $pc' + 1 = \text{fst}(\text{seg}_{BIR}(m, pc' + 1))$. That is, $pc' + 1$ is the first control address in the next code segment, after the current segment delimited by the $\text{invo}se$ special instruction.

The next set of edges depends on the presence of an exception handler for the NullPointerException. If there is a suitable handler, then there is another edge $\phi_m^p \rightarrow \phi_m'^{N.P.E., R}$, which is matched by $\phi_m^C \rightarrow \phi_m'^{C'}$, which traverses all the nodes extracted from $H\text{Save}$, and $(\phi_m^C, \phi_m'^{C'}) \in R$ since $\text{seg}_{BC}(m, \text{suc}(p)) = \text{seg}_{BIR}(m, pc')$ and $pc' \in \text{op}(s, \{N.P.E.\})$. Also, there is the edge $\phi_m^p \rightarrow \phi_m'^{N.P.E., R}$, matched by $\phi_m'^{C', N.P.E.} \rightarrow \phi_m'^{C'}$, and again $(\phi_m'^{C'}) \in R$ since $\text{seg}_{BC}(m, q) = \text{seg}_{BIR}(m, pc' + 1)$. If there is no handler, there is the edge $\phi_m^p \rightarrow \phi_m'^{p, N.P.E., R}$, which is matched by $\phi_m^C \rightarrow \phi_m'^{C'}$, which traverses all the nodes extracted from $H\text{Save}$, and $(\phi_m^C, \phi_m'^{C'}) \in R$ since $\text{seg}_{BC}(m, p) = \text{seg}_{BIR}(m, pc')$ and $pc' \in \text{op}(s, \{N.P.E.\})$.

The set of edges extracted for the propagation of an exception $x$ also depends on the presence of a handler, and the explanation is similar to the one for NullableException. If there is a suitable handler, then there is another edge $\phi_m^p \rightarrow \phi_m'^{N.P.E., R}$, which is matched by $\phi_m^C \rightarrow \phi_m'^{C'}$, which traverses all the nodes extracted from $H\text{Save}$, and $(\phi_m^C, \phi_m'^{C'}) \in R$ since $\text{seg}_{BC}(m, p) = \text{seg}_{BIR}(m, pc')$ and $pc' \in \text{op}(s, \{x\})$. Also, there is the edge $\phi_m^p \rightarrow \phi_m'^{N.P.E., R}$, which is matched by $\phi_m'^{C', N.P.E.} \rightarrow \phi_m'^{C'}$, and also $(\phi_m'^{C'}) \in R$ since $\text{seg}_{BC}(m, t) = \text{seg}_{BIR}(m, pc')$. If there is no handler, there is the edge $\phi_m^p \rightarrow \phi_m'^{p, x, r}$, which is matched by $\phi_m^C \rightarrow \phi_m'^{C'}$, which traverses all the nodes extracted from $H\text{Save}$, and also $(\phi_m'^{C'}) \in R$. Thus $(\phi_m'^{p, x, r}, \phi_m'^{p, x, r}) \in R$ since $\text{seg}_{BC}(m, p) = \text{seg}_{BIR}(m, pc')$. If there is no handler, then $pc' = \text{fst}(\text{seg}_{BIR}(m, pc'))$. Notice that if propagated exception $x = N.P.E.$, then $pc'$ is the only possible control address tagging the node with $XP = \{N.P.E.\}$, which has been shown above to be in $R$ for either the cases of having or not a handler.
Subcase II The receiver of a call from invokespecial is a method within the same class, or from the super class. The BC2BIR\textsubscript{instr} transformation returns the [notnull] instruction, followed by a sequence of assignments (denoted by HSave), and the invocation instruction:

\[ \text{BC2BIR}_{\text{instr}}^{\text{null}} = \text{[notnull; HSave(pc, as); } t_\text{x} := \text{e.n(...)}] \]

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

\[ g_{\text{fin}}^{m, \text{pc}[\text{notnull}]} = \begin{cases} 
\{ \alpha \}_{m, \text{pc}+1} \to \alpha_{m} & \text{there is a handler} \\
\{ \alpha \}_{m, \text{pc}+1} \to \alpha_{m} & \text{otherwise}
\end{cases} \]

As explained above, the extraction of the assignments from the HSave function produces a (path-like) graph corresponding to a weak transition \( \alpha_{m}^{p} \to \alpha_{m}^{t} \).

\[ g_{\text{fin}}^{m, \text{pc}[\text{null}]} = \begin{cases} 
\{ \alpha_{m}^{p} \cap \alpha_{m}^{t} \} \cup N_{m}^{p, n} & \text{there is a handler} \\
\{ \alpha_{m}^{p} \cap \alpha_{m}^{t} \} \cup N_{m}^{p, n} & \text{otherwise}
\end{cases} \]

Then \( \alpha_{m}^{p} \cap \alpha_{m}^{t} \), which traverses \( \alpha_{m}^{p} \to \alpha_{m}^{t} \), and the nodes extracted from HSave. Thus \( (\alpha_{m}^{p}, \alpha_{m}^{q}) \in R \) since \( \text{seg}_{\text{IBC}}(m, \text{suc}(p)) = \text{seg}_{\text{BIR}}(m, \text{pc}^{+1}) \) and \( \text{fsc} \left( \text{seg}_{\text{BIR}}(m, \text{pc}^{+1}) \right) = \text{pc}^{+1} \).

The set of edges extracted because of a potential NullPointerException depends on the presence of a handler. If there is a suitable handler, then there is an edge \( \alpha_{m}^{p} \to \alpha_{m}^{p} \to \alpha_{m}^{p} \), which is matched by \( \alpha_{m}^{p} \to \alpha_{m}^{p} \to \alpha_{m}^{p} \), and \( \alpha_{m}^{p} \to \alpha_{m}^{p} \) is \( \text{pc} \) since \( \text{seg}_{\text{IBC}}(m, \text{pc}) = \text{seg}_{\text{BIR}}(m, \text{pc}) \) and \( \text{pc} \in \text{op}(s, \text{N.P.E.}) \).

Also, there is an edge \( \alpha_{m}^{p} \to \alpha_{m}^{p} \), which is matched by \( \alpha_{m}^{p} \to \alpha_{m}^{p} \), and \( \alpha_{m}^{p} \to \alpha_{m}^{p} \) is \( \text{pc} \) since \( \text{seg}_{\text{IBC}}(m, \text{pc}) = \text{seg}_{\text{BIR}}(m, \text{pc}) \) and \( \text{pc} \in \text{op}(s, \text{N.P.E.}) \). Notice that in either of the cases \( \text{pc} \) may not be the only control address tagging a node with N.P.E., and \( \alpha_{m}^{p} \), or \( \alpha_{m}^{p} \), must also relate to such node.

Again, the explanation for edges related to a propagated exception \( x \) is similar to the one for N.P.E.. If there is a handler for \( x \), then there is an edge \( \alpha_{m}^{p} \to \alpha_{m}^{p} \),
APPENDIX A. CORRECTNESS OF \( \mathcal{G}_{\text{BIR}} \circ \text{BC2BIR} \)

\( \bullet^{p,x}_m \), which is matched by \( o^p_m \xrightarrow{n} o^{pc'+x}_m \), which traverses \( o^{pc+1}_m \) and the nodes extracted from \( H_{\text{Save}} \), and \( (\bullet^{p,x}_m, \bullet^{pc'+x}_m) \in R \) since \( pc' \in oap(s, \{x\}) \). There is also an edge \( \bullet^{p,x}_m \xrightarrow{\cdot} o^t_m \) which is matched by \( \bullet^{pc'+x}_m \xrightarrow{\cdot} o^{pc}_t \), and also \( (\bullet^{t,x}_m, \bullet^{pc'+x}_m) \in R \) since \( seg_{\text{JBC}}(p, t) = seg_{\text{BIR}}(m, pc') \) and \( pc' = \text{fst}(seg_{\text{BIR}}(m, pc')) \). If there is no handler, then there is an edge \( \bullet^{p,x}_m \xrightarrow{\cdot} \bullet^{p,x,r}_m \) that is matched by \( \bullet^{pc}_m \xrightarrow{\cdot} \bullet^{pc',x,r}_m \), and \( (\bullet^{p,x,r}_m, \bullet^{pc',x,r}_m) \in R \) since \( \text{seg}_{\text{JBC}}(p, r) = \text{seg}_{\text{BIR}}(m, pc') \) and \( pc' \in oap(s, \{x, r\}) \).

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

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

\[
\begin{align*}
\text{BC2BIR}_{\text{Instr}}^{\text{return}} &= \text{[return expr]} \\
\mathcal{G}_{\text{BIR}}^{m, p, \text{[return expr]}} &= \{ \tau^p_m \xrightarrow{\cdot} o^p_m \}
\end{align*}
\]

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

\( \square \)
Appendix B

Correctness of oG

We now present the full definitions and results that have been summarized in Section 4.3. For this, once again we use the notion of open environment (Figure 4.1) to model incomplete Java bytecode programs.

The goal of reasoning about open environments is to analyze the available code, and produce results that hold for any complete system that implements the incomplete system. Thus, we need to set constraints for instantiating the unavailable components, so that properties established over the original incomplete system are preserved once the system becomes complete. The arrival of a new component redefines an open environment, defined by the previous components, plus the newly added component.

Now we formally define the constraints for property preservation presented in Definition 15 as a refinement pre-order on open environments.

Definition (Environment Refinement). Let $\Gamma_o$ and $\Gamma_o'$ be two open Java bytecode environments, and $\Gamma$ be a closed environment. We define the refinement relation $\preceq$ between environments with the rules in Figure B.1. We say that $\Gamma_o$ refines $\Gamma_o'$ if $\Gamma_o \preceq \Gamma_o'$. Also, we say that $\Gamma$ implements $\Gamma_o$ if $\Gamma \preceq \Gamma_o$.

The refinement between two open environments is relativized on the refinement of the partial mappings representing the classes, methods and interfaces. Obviously the domains for each of the partial mappings in both environments must be the same. This is a necessary constraint to ensure that the open environments are defined for the same set of components.

An interface $\Gamma_I^I[\omega]$ refines $\Gamma_I^I'[\omega]$ if both environments provide the same methods, and if the new interface inherits a subset of set of the interfaces inherited by the old one. These requirements are essential to guarantee that the virtual method call resolution produces a sound over-approximation of the possible call receivers.

A class $\Gamma_C^C[\sigma]$ refines $\Gamma_C^C'[\sigma]$ if it fulfills three requirements. First, the super class must be the same in both environments. The second requirement is that $\Gamma_C^C'[\sigma]$ implements a subset of the interfaces implemented by $\Gamma_C^C[\sigma]$. This is a relaxation
from the definitions, and it is allowed because implementing fewer interfaces implies that the original class is an over-approximation of the refined class. There are no requirements to fields since they do not have relation to the class hierarchy.

The refinement between two methods $\Gamma_o^M[m]$ and $\Gamma_o'[m]$ is defined for three distinct cases. The first is between two methods with empty bodies. In this case, $\Gamma_o^M[m]$ is said to refine $\Gamma_o'[m]$ if the set of exceptions that the former guarantees not to propagate is a superset of the exceptions not propagated by the latter. 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 $\Gamma_o^M[m]$ and $\Gamma_o'[m]$ must have the same code and handlers. The third, and most relevant case, is the one where an empty method $\Gamma_o^M[m]$ is refined by a concrete method $\Gamma_o'[m]$. Let $EXCP$ be an auxiliary function that returns the set of exceptions that a method 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.

One important property of the definition of open environments for Java bytecode is that the refinement relation preserves the type hierarchy (Definition 12 on page 54). This means that the subtyping relation between classes is the same, and the subtyping relation between interfaces, and interfaces and classes in the original open environment is an over-approximation of the hierarchy in the refined model.
A consequence of type hierarchy preservation is that the set of exceptions is also known (Definition 13, page 55). Since the class definitions are equivalent for both the original environment, and for the open environment that refines it, the monotonicity of the exception types is preserved.

Now we enunciate Theorem 3 once again, and present the proof of monotonicity of the refinement relation w.r.t. set inclusion of CFGs, and consequently also w.r.t. structural simulation.

**Theorem** (Containment of CFGs). Let \( \Gamma_o \) and \( \Gamma_o' \) be open environments, and \( m \) be the signature of a method available in both environments. Then \( \Gamma_o \preceq \Gamma_o' \) implies \( oG(m, \Gamma_o) \subseteq oG(m, \Gamma_o') \).

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

By the hypothesis, method \( m \) is implemented in both environments. Also by the hypothesis, \( \Gamma_o \preceq \Gamma_o' \) holds. Thus, from the refinement definition (Figure B.1, on page 110), the instruction arrays and exception tables are the same (i.e., \( \Gamma_o^M[m].\text{code} = \Gamma_o'^M[m].\text{code} \) and \( \Gamma_o^M[m].\text{handler} = \Gamma_o'^M[m].\text{handler} \)).

From the definition of function \( oG \) (Figure 4.2 on page 56) one can see that it yields the same set of triples for all instruction groups whenever two methods have the same instruction array and exception table, except for \( \text{NewObject} \) and \( \text{MethodCall} \). Thus, the trivial cases are proven, and there are only two cases left.

The sets \( \text{NewObject} \) and \( \text{MethodCall} \) 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 of a virtual method call, however, depend on the class hierarchy \( L \) of each open environment. Moreover, the computation of \( N_{pc}^m \) also depends on the definitions of the open environment. Therefore, we present the proof for the virtual case of \( \text{MethodCall} \). The other are simpler cases, where there is a single receiver for the call, and the proof is analogous.

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

Let \( MCA_L(C, \text{ns}) \) and \( MCA_L(C, \text{ns}) \) be the sets of all possible receivers of the same virtual method call in the original and in the refined open environments, respectively. They are defined as all the methods of subtypes of \( C \) 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 \( \Gamma_o^C \) and \( \Gamma_o'^C \) have the same super class. The subtyping relation (Figure 2.5, page 17) defines that classes
APPENDIX B. CORRECTNESS OF $\mathcal{G}$

can only be subtyped by another class. Thus, the set of subtypes of $C$ is the same in both environments, and $MCA_{\mathcal{G}}(C.ns) = MCA_{\mathcal{G}}(C.ns)$.

Now let $C$ be an interface. Its subtypes are all the classes (and their subtypes) that implement the interface, and its sub-interfaces. The refinement relation defines that classes in the refined open environment must implement a subset of the interfaces ($\Gamma_{o}[\sigma].\text{interfaces}$) and its sub-interfaces ($\Gamma_{o}[\omega].\text{interfaces}$), as defined in the original environment. Thus, the subtypes of $C$ in the refined environment are a subset of subtypes of $C$ in the original environment. Therefore, $MCA_{\mathcal{G}}(C.ns) \subseteq MCA_{\mathcal{G}}(C.ns)$.

Next, we show that the set of propagated exceptions by an arbitrary receiver of a method call in $\Gamma_{o}[m].\text{code}$ is a superset of the exceptions propagated by the same call in $\Gamma_{o}[m].\text{code}$. Let $\Gamma_{o}[n]$ be one possible receiver for a method invocation within $\Gamma_{o}[m]$. 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 $N_{\text{pc}}^\mathcal{G}$ 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 ($\text{Any} - \Gamma_{o}[n].\text{handlers}$). Therefore, the number of edges produced by $N_{\text{pc}}^\mathcal{G}$ 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 unavailable 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, since the refinement relation holds between the two environments (by the hypothesis), 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 both available 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 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 by an invoked method, are both subsets in the refined environment, when compared to the original environment. Therefore, as shown for all cases, the set of produced edges is also a subset.

As mentioned in Section 4.2, a closed environment is a special case of an open environment, where there are no missing methods. Moreover, the CFG extracted modularly from an open environment contains the CFG from any complete pro-
gram that implements the open environment. Now we connect these facts with the soundness of the extraction algorithm for complete programs.

We show that the CFGs extracted with $oG$, and those extracted using $G_{\text{MCA}}$ with $MCA$ as the VMC algorithm are the same for complete programs. Therefore, CFGs extracted with $oG$ structurally simulate the ones extracted with $G_{\text{MCA}}$, and we refer once again to [35] to conclude behavioural simulation, following the argument presented in Appendix A.

We now restate Theorem 4 and present the proof.

**Theorem (CFG equality).** Let $\Gamma$ be a closed environment, and $G_{\text{MCA}}$ be the instantiation of $G_{\text{MCA}}$ with $MCA$. Then $G_{\text{MCA}}(\Gamma) = oG(\Gamma)$.

**Proof.** The proof follows from the fact that if there are no missing components, then the function $N_{\text{PC}}^c$ 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 for both algorithms. Therefore, the extraction rules for $oG$ are reduced to exactly the same extraction rules as for $G_{\text{MCA}}$. \qed
Appendix C

Open Environment Storage File

An open environment, which models an incomplete Java bytecode program, is stored in a unique XML file, as described in Section 3.3. The XML files also store some other partial results obtained during an execution of CONFLEx. The data structure for the open environment is defined in a separate Document type definition (DTD) file, shown in Figure C.1. CONFLEx validates the format of the XML file against the DTD specification upon loading.
APPENDIX C. OPEN ENVIRONMENT STORAGE FILE

![Figure C.1: DTD file describing the data structure for result storage](image-url)
Appendix D

CPN for Producer / Consumer

Figure D.1 presents the complete CPN extracted from the SyncTask program in Figure 5.3 (page 70). As explained in Section 5.1, the hierarchical version is composed of twenty-one subpages. This leads to lots of redundant graphical elements, e.g., many instantiations of fusion places, and the layout becomes cumbersome. Thus, we present the non-hierarchical version of the CPN, which provides a more compact and intuitive representation. However, it is important to stress that the hierarchical and non-hierarchical representations of CPNs are semantically equivalent.

We generate the non-hierarchical net in Figure D.1 by replacing syntactically the STs with their respective subpages. That is, we expand a subpage inside its parent page, and collapse the in-ports and in-sockets, and out-ports and out-sockets. Notice that the inverse process of moving sup-parts of a CPN to subpages, and representing them with STs is also possible. We preserve the notation for fusion places to help the reader identifying these places in the hierarchical model. However, we typically collapse all instantiations of a fusion place into a single normal place in a non-hierarchical net.
Figure D.1: Non-hierarchical Coloured Petri Net extracted from SyncTask