Master's Thesis

Verification of Java Programs with Exceptions with CPAchecker

Chair of Software and Computational Systems Lab
Ludwig-Maximilians-Universität München

Benedikt Damböck

Munich, 11 29th, 2023

Submitted in partial fulfillment of the requirements for the degree of M. Sc.
Mentor Dr. Philipp Wendler
Supervised by Prof. Dr. Dirk Beyer

Acknowledgement

I would like to thank Prof. Dr. Dirk Beyer for giving me the opportunity to write the master's thesis at the chair for Software and Computational Systems Lab. Special thanks goes to my mentor Dr. Philipp Wendler. I am grateful for his guidance throughout the different steps that come with writing a master's thesis. The weekly meetings and fast answers when i had a question helped me a lot. I would also like to thank my family, friends and especially my partner for their support. They cheered me up whenever I felt frustrated and celebrated the moments of success with me.

Abstract

Errors can occur in Java programs that must be handled by Java's exceptional control flow. Verification tools that provide support for Java programs must be able to handle these exceptions. CPAchecker is one of these tools, but it is not able to properly analyze programs with exceptions in them. It uses control-flow automata to represent the program internally and performs its analysis on this structure.
In this thesis, we look at approaches that other tools use to handle exceptions in Java and talk about an approach to add the exceptional control flow to the control-flow automata by using standard non-exceptional Java control flow. A global variable is being used to keep track of an exception that is actively affecting the program. Exception handling is represented by conditional statements that check whether an exception has occurred and whether it can be handled. We implement this approach in CPAchecker and compare it with 8 state-of-the-art Java verification tools. All programs in the dataset that contained developer-thrown exceptions could be analyzed. However, situations with abnormal execution and exceptions thrown by a library couldn't be properly analyzed. The thesis shows that this approach has improved CPAchecker's ability to analyze Java programs, but that there are still some issues that need to be addressed before it can perform as well as other state-of-the-art tools.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Software is one of the most important cornerstones of modern society. There are many different parts of our lives that improve every day thanks to advances in hardware and software. The one thing that all the different software applications have in common is that they must work reliably. This requirement has given rise to two new areas of research: software testing and software verification. Software verification isn't as popular as software testing in the enterprise sector, except for companies that produce safety-critical software. Agencies such as NASA and universities advance the field steadily. One of the problems these verification tools have to analyze is erroneous behavior in the program control flow. There are many different verification tools for popular programming languages that offer different approaches to do so. Java, being one of the most popular programming languages, needs tools that can handle all the features that Java offers, including exceptions.
In this paper, two different topics will be discussed. An approach to include exception handling in a control flow automaton using standard Java control flow is being presented. A global static variable is used to track the exception that is actively affecting the program, as well as conditional statements that check whether an exception can be handled. The advantage of this approach is that the exception handling mechanism only has to be introduced at the control flow automaton creation and does not have to be dealt with individually in each analysis. The typical exception handling constructs and an example of an abnormal execution will be discussed. The other topic is a review of the current state of exception handling in verification and adjacent fields. There are many different approaches, ranging from using a local variable to track an exception, to handling exceptions in the analysis itself, to using a listener on a Java virtual machine.
The approach discussed in this thesis is implemented in the software verification tool CPAchecker, which provides Java verification support but is currently unable to analyze programs with exceptions. The implementation is evaluated to see if the accuracy of CPAchecker is improved, by using value analysis in conjunction with runtime type analysis on a set of programs. At the end CPAchecker will be compared to other available Java verification tools that participated in the SV-COMP 2023[1] software verification competition.

[1] `https://sv-comp.sosy-lab.org/2023/`

2 Background

2.1 Software Verification

Software verification uses formal verification to verify that a software system is correct. The article „What is formal Verification?“[11] defines formal verification as the use of mathematical methods to prove that a well-defined notion of functional correctness applies to a design. To obtain a proof of correctness about a computer program, two different approaches are being used as discussed in the paper „Formal software verification: Model checking and theorem proving“[26]. The first one is the pre/post condition, where a problem is being defined as a relation between a formula that is assumed to hold at the beginning of program execution and a formula that is assumed to hold at the end of the program execution. A program is correct if the post condition formula holds, given the pre-condition formula. The second approach is to use invariant assertions. For this approach an invariant formula must be set up and must hold throughout the execution of the program.

2.2 Control-Flow Automaton

A control-flow automaton (CFA) is a directed graph that represents the control flow of a program. It is represented by a triple (L,l_0,G). L is the set of program locations, l_0 is the starting location and G is the set of edges between the locations in L. An edge $g = (l, op, l') \in G$ consists of the location where the edge starts $l \in L$, the location the where the edge ends $l' \in L$ and the operation $op \in Ops$. Ops contains assumptions, declarations and statements.

2.3 Explicit Value Analysis

Explicit value analysis [10] uses the values of all fields in a program to determine the correctness of the program. The analysis checks whether the values that are given at any point in the program can violate a given specification. If there is no path that violates the specification the program is considered safe. Value analysis considers only equalities and non-equalities when analyzing a program. If a value is not present, it is considered to be an unknown that can still be used in conditional statements. All possible branches must be explored when encountering an unknown value in a branching statement.

2.4 Runtime Type Analysis

Runtime Type Analysis (RTT) is an analysis that tracks the concrete runtime types of all objects. It works similarly to the Explicit Value Analysis.

2.5 Java Assertion

An assertion in Java is a statement consisting of the assert keyword and a boolean expression. The „Java SE Documentation“[3] defines it as a statement that is being used to test assumptions about code. A program containing an assertion statement runs

correctly if the boolean expression evaluates to true, and throws an error if it evaluates to false when assertions are enabled. Another expression can be added to the assertion statement: assert $expr_1$: $expr_2$. The first expression, $expr_1$, is a boolean expression, and the second, $expr_2$, is an expression that has a value that is being used as the error message.

2.6 Guarded Command

A guarded command is a statement or a list of statements that is executed when a condition is fulfilled. This construct was first introduced by Dijkstra [19] in 1975.

2.7 Verification Condition

Frade and Pinto [20] define verification conditions (VC) as first-order proof obligations. Whether a verification condition is valid or invalid can be determined using standard proof tools. A program can be considered correct if all verification conditions are proven to be valid.

2.8 Exception Handling in Java

Java exceptions are a mechanism for indicating that a problem has occurred that could lead to an error in the execution of a program. The authors of „How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications“[13] define Java exception handling as reliability-driven exception handling. This means that the software reliability can be improved by having to explicitly define exception handling constraints.
The Java language specification for exceptions [2] explains that the developer will be warned by the Java virtual machine by throwing an exception when a semantic constraint is being violated. In Java, when an exceptional circumstance happens, it is called throwing an exception. When an exception encounters a construct that can handle it, it is called catching an exception. A developer can indicate that an exception occurs in his code by using the `throw` statement, which consists of the `throw` keyword and an expression. The expression can be either a value of the type `Throwable` or one of its subclasses, or a `null` reference. If the expression is neither, a compile-time error occurs. After an exception is thrown, all statements are ignored until a `try` statement is encountered that contains a matching `catch` clause. The thread to which the `throw` statement belongs is killed if no such `try` statement exists. A method or constructor declaration can contain a `throws` clause that indicates that it potentially throws an exception but doesn't handle it itself.
There are two types of exception classes: the unchecked exception classes include the runtime exception classes and the error classes, while the checked exception classes include `Throwable` and all of its subclasses except for the unchecked exception classes. There are three different ways to trigger the exception mechanism: when the Java virtual machine executes a `throw` statement, when an abnormal execution condition is detected, or when an asynchronous exception occurs. An abnormal execution occurs when the normal semantics of the Java programming language is violated, when loading, linking, or initializing a section of the program results in an error, or when the Java virtual machine cannot

continue due to an internal error or resource limitation. Asynchronous exceptions can also occur when the Java virtual machine cannot continue in a concurrent program, and when the stop method of the `Thread` or `ThreadGroup` class is used.
The `try-catch(-finally)` construct is used to handle exceptions in Java. A `try` statement encloses a block of code. When this code is executed and an exception is thrown, the `catch` clauses are checked if one matches the exception, if the `try` statement has `catch` clauses. If a `catch` block matches the exception, the control flow will be continued at the `catch` clause and the code that is responsible for throwing the exception does not get executed. If more than one catch clause matches the exception, only the first one will be executed. If none of the `catch` clauses match the exception, the next closest `catch` clause that can handle the exception will be executed. A thread is terminated if an exception isn't caught in that thread. If there is a `finally` clause, the block of code enclosed by the finally clause is executed after the `try` statement and the `catch` clause, even if an exception is thrown in the `try` block or a `catch` clause gets executed. The `catch` clause consists of the `catch` keyword and a CatchFormalParameter, which consists of a CatchType that is a ClassType, a VariableDeclaratorId, and optionally of a VariableModifier. The following figures Figure 1, Figure 2, and Figure 3 are examples that show some paths that an exception can take. They are based on Listing 1, which is written in a Java-style pseudocode form.

```
1  void exceptionMethod(){
2      try{
3          foo(); \\may throw an exception e
4          bar(); \\exception free method
5      } catch(Exception e) {
6          c \\catch block
7      } finally {
8          f \\finally block
9      }
10     r \\rest of method
11 }
```

Listing 1: Exception example program

2.9 CPAchecker

CPAchecker [9] is an open source configurable software verification tool developed in Java for C and Java programs. It offers the possibility to use different program analysis approaches and model checking approaches.
The central data structure used to represent a program in CPAchecker is a set of CFAs. The analyses that can be applied to Java programs are value analysis and runtime type analysis. CPAchecker provides a framework that is called configurable program analysis for introducing new analyses. It currently does not provide support for Java programs with exceptions. The exceptional control flow is being ignored, but the content of the code blocks within the try statement and the catch and finally clauses are added to the CFA. This can lead to the analysis returning a correct result by chance. The CFA in Figure 4 is generated when analyzing the program that contains an exception and an exception handling construct in Listing 2 with the current CPAchecker version[2].

[2] `https://svn.sosy-lab.org/software/cpachecker/trunk/?p=45346`

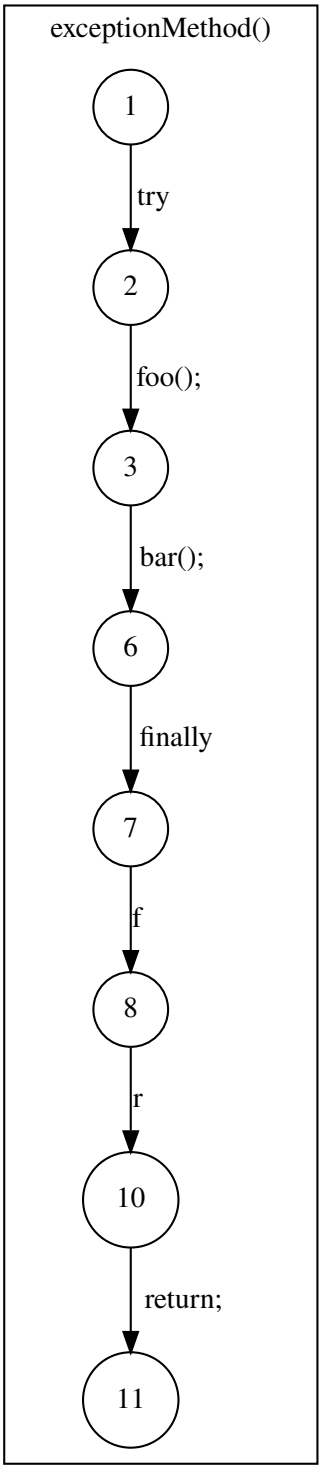

Figure 1: Control flow - exception not thrown

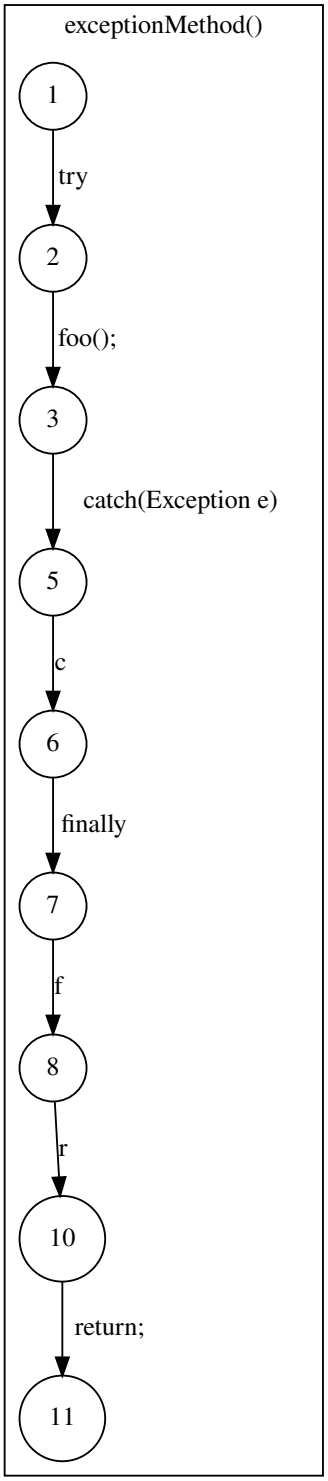

Figure 2: Control flow - exception thrown and caught

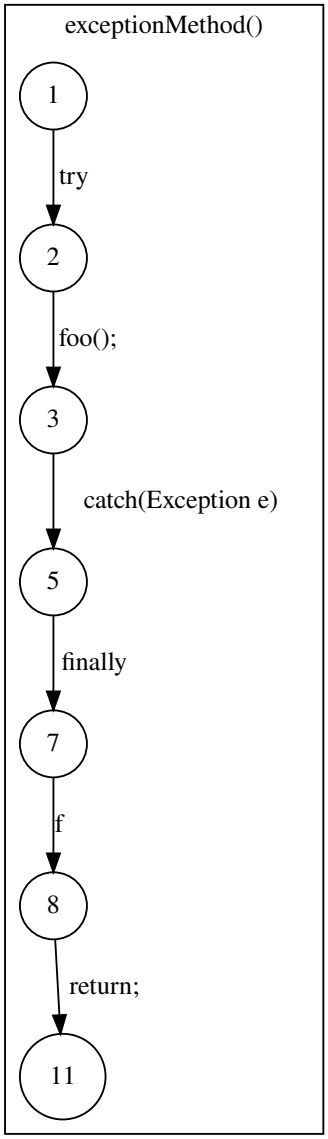

Figure 3: Control Flow - exception thrown and not caught

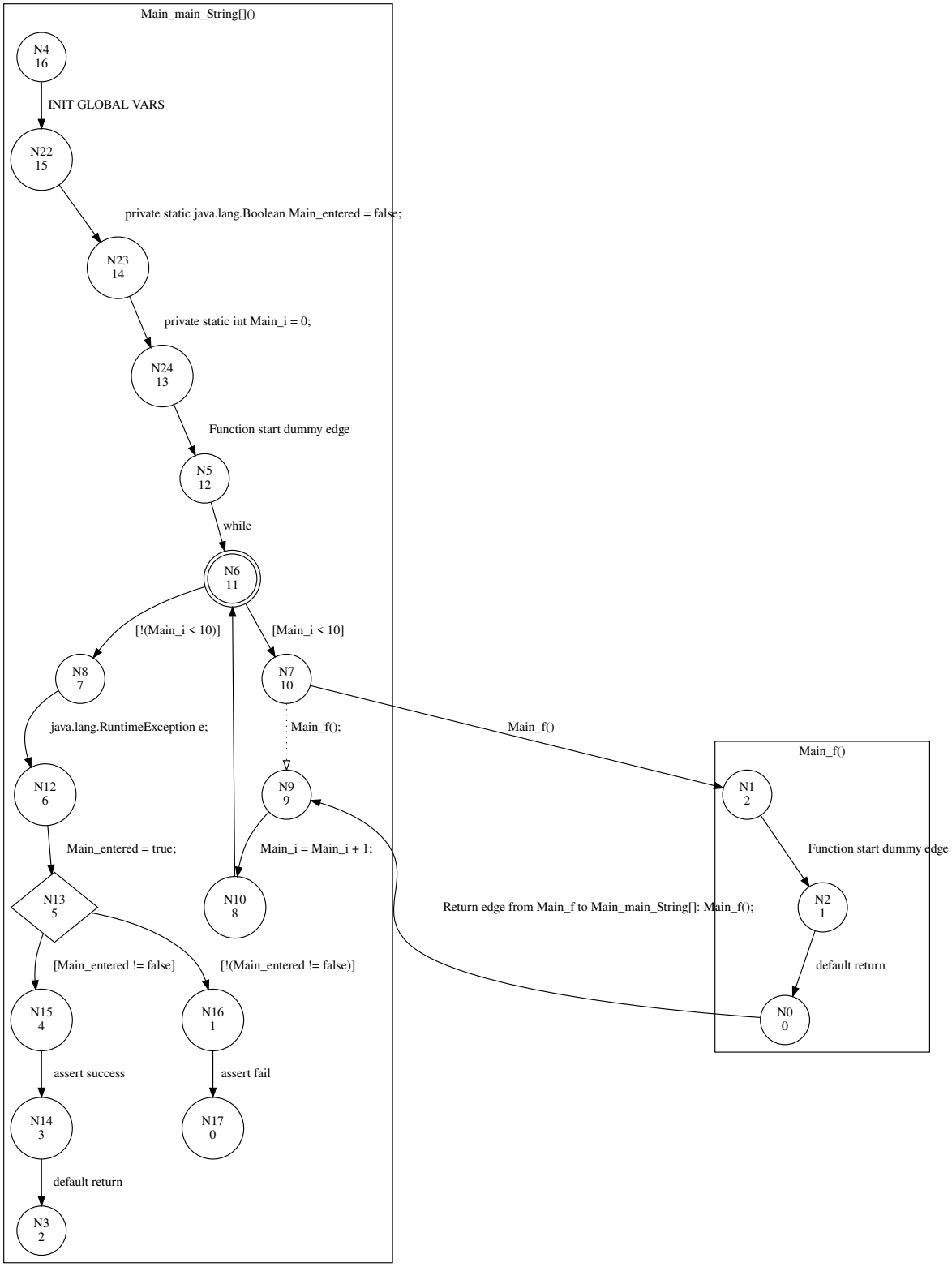

Figure 4: Original program included exception, but exception control flow not included in CFA

```java
public class Main {
    private static int i = 0;
    private static Boolean entered = false;

    private static void f() {
        throw new RuntimeException();
    }

    public static void main(String[] args) {
        try {
            while (i < 10) {
                f();
                i++;
            }
        } catch (RuntimeException e) {
            entered = true;
        }
        assert entered;
    }
}
```

Listing 2: Programm with exception correctly solvable by CPAchecker

2.10 Configurable Program Analysis

A configurable program analysis (CPA) [8] allows the abstract domain specification to be separated from the analysis algorithm itself. The CPA algorithm is a reachability analysis algorithm that can be used with any CPA. Multiple CPAs can be combined to a composition of CPAs.

3 Related Work

3.1 Helper Variable based Exception Analysis

A possible technique used to describe exceptions for an analysis of a program is the use of a helper variable that handles the representation of the variable in the respective format, e.g. directed graph. JayHorn [4][1][23] is one of the tools using this technique. It is a software model checking tool for Java that can analyze programs that do not contain multithreading or dynamic class loading. It uses the Soot framework, which parses the code and returns a JIMPLE representation, which is a simplified version of Java source code. The Jimple representation is translated into a set of Horn clauses, which then get analyzed by a Horn engine. To handle exceptional control flow, a static variable is introduced that keeps track of the last exception thrown. This static variable is getting checked every time the control flow changes, e.g. when a method is getting called.
JBMC [17][18] is another verification tool for Java byte code. The code is translated into a static single assignment form that includes a control flow graph. When an exception occurs in the code, it is stored in a global variable. The control flow from the exception case is being written as conditional branches. JBMC will attempt to match the exception with a catch statement in this exceptional control flow. The GOTO program resulting from this translation is then analyzed by a bounded model checker and checked for unwanted behavior. A solver or configured SAT is then used to solve the bit-vector formula from the previous process. One of the features of JBMC is that it is able to detect both runtime and user-specified exceptions. It creates safety conditions that check for various exceptional behaviors such as null dereference or array bounds errors.
Translation of Annotated Code (TACO)[16] is a SAT-based tool that uses bounded verification to analyze sequential Java programs. The Java code must be annotated with either the Java Modeling Language or JFSL. It translates the annotation and the code into SAT formulas using intermediate languages such as JDynAlloy. One of the problems of this concept is the lack of exception management in JDynAlloy. The solution to this problem is the introduction of a new variable which represents the exception and can be analyzed as such.
In „Interprocedural Exception Analysis for C++“[27], a framework is discussed that introduces interprocedural exception analysis and a method for transformating C++ programs that contain exceptions into exception-free programs. The transformation process is divided into three steps. Transformation of the C++ code into intraprocedural exception control flow graphs. An exceptional return is being introduced for each function in addition to the normal return. A new edge type is introduced: the exception edge. The content of the edge type is a Signed-TypeSet containing either a plus or a minus sign and a set of exception types. A TypeSet containing a plus sign signals that the exception types in the set are supposed to be included in the analysis. Including a minus in the TypeSet signals that the condition is met if it is an exception type other than the types in the set. The next step consists of combining these graphs into a interprocedural exception control flow graph. An exception analysis is used to refine the information on the exception edges in the interprocedural exception control flow graph. In the last step, an exception-free program is generated. This consists of introducing a type-id helper variable that holds the type that is being thrown and local exception-objects that represent each exception

type that the function can throw. The functions parameter is expanded by a reference parameter that holds the type-id and a reference parameter for each exception that is thrown in the function. A throws keyword is replaced by assigning the exception type to the helper variable. The catch construct is represented by a switch statement on the local type id. A GO-TO statement is added that leads either to the content of a catch block or the exceptional-exit node.

3.2 Java Virtual Machine Listener

One way to handle exceptions in software verification is to use a listener on a virtual machine. The information that the listener encounters can be saved and reused later.
Java Pathfinder (JPF) [21] and its various extensions use this technique. JPF uses a custom virtual machine that can store and match program states. JPF's main feature is the ability to backtrack to previously explored states. It can handle concurrent programs and is able to check for various properties such as runtime errors, assertion violations, and deadlocks. It uses an explicit-state model checker to analyze a given problem.
Symbolic Pathfinder (SPF) [28][24] is an extension to JPF that introduces symbolic execution of Java byte code. It implements a byte-code instruction set that uses symbolic execution principles. An important feature is the ability to enable it at any point of the program analysis. The main goal of SPF is automatic test case generation. The authors implemented a custom listener class that listens for exceptions and adds all of the necessary information to a summary if it is a runtime exception. It adds a warning for every runtime exception encountered even if it's handled by the developer, because the authors believe that relying on runtime exceptions in code is bad practice.
JDart [25] is another JPF extension that uses dynamic symbolic analysis on Java programs. It's modular and has an executor and an explorer at its core. The executor runs the program with different concrete data values and records symbolic constraints for the different paths it follows. The explorer organizes a constraint tree while also handling termination, exploration strategies and support for the constraint solver. This core can be extended with extensions like JUnit Test Generator. Paths that lead to an ERROR state containing unhandled exceptions are added to the constraint tree along with their value. It tracks the number of unique satisfiable paths, along with their termination state (OK or ERROR).

3.3 UML based Exception Analysis

A technique that falls under the category of architectural verification is UML activity graph-based exception verification. In the paper „Architecting fault tolerance with exception handling: Verification and validation“[12] a fault-tolerant architectural element is presented: Idealised Fault-Tolerant Architectural Element (iFTE). It provides multiple interfaces to depict normal and abnormal behavior. The Provided Abnormal interface is responsible for signaling the exceptions a class may throw, and the Required Abnormal interface indicates, which exceptions can be handled by the class. A thrown exception that is not part of the Required Abnormal interface of a class, leads to an internal iFTE error, which produces a failure exception. Each abnormal use case requires a UML component diagram containing

the interfaces and a set of UML sequence diagrams containing abnormal scenarios. The UML files are translated into a formal notation using B-Method, a formal language, which is supported by Communicating Sequential Process, an algebraic process to present execution sequences.

3.4 Constraint Based Exception Analysis

The authors of „Interprocedural Exception Analysis for Java„[15] discuss a formal verification method for finding unhandled exceptions. The approach they discuss is an improvement to the intraprocedural exception analysis of the current JDK Java compiler, which uses only the developer's specification. The interprocedural exception analysis reports unnecessary try-catch constructs, as well as improving existing catches in the form of specifying the caught exception when, for example, a parent class is used. A set-constraint framework is used to analyze classes. Chang et al. discuss the analysis at two different levels: expression-level and method-level. At the expression-level each object class of each expression generates a set constraint. This wastes a lot of resources because exception statements are usually rare. Method-level analysis only considers set variables for class methods and try blocks, while preserving the information that you would get from expression-level analysis.
Another constraint-based approach used in the tool Jex is discussed by Robilliard and Murphy [29]. The tool parses a Java source file and generates an abstract syntax tree. The AST visitor is responsible for storing try-catch-finally constructs and generating a set of exception types for exceptions that are explicitly thrown by the developer, thrown by a system operation (e.g. division by zero), or propagated by a method. Each statement is checked to generate the set of exception types. The information is then analyzed to generate a Jex file that contains a description of the exception flow in a class.

3.5 Relation Based Exception Analysis using a Directed Graph

In a relation based exception analysis approach the exception handling constructs are added to a directed graph and an analysis has to handle the way the exceptions are approached. The authors of „Static analysis of exception handling in Ada“[31] discuss an approach for Ada programs. They present a four-step automatic process for analyzing exception constructs in Ada programs. A semantically correct Ada source file is compiled into a non-optimized intermediate form called DIANA. While analyzing the DIANA output, a directed graph is built that contains all of the exception information. In the third step, the directed graph is used to define relations that are useful for exception handling analysis, and these are written to a separate text file. In the next step, the relations defined previously are used to identify violations of developer-defined guidelines.

3.6 Property Based Exception Analysis

An architectural exception flow analysis is being discussed in the paper „Specification of exception flow in software architectures„[14]. They use ACME as a software architecture description language and Aereal to describe and analyze the exception control flow in a program. The specification process, which is part of software architecture process in this

case, includes failure scenarios, exceptions that handle these scenarios, and the specific way in which these exceptions handle the failures, among other things. Aereal includes point-to-point connectors that are used to describe exceptions between software components. The connectors are unidirectional. The Aereal specification, which consists of a set of ACME systems and family descriptions, is then converted into an Alloy specification. The analysis checks whether the resulting Alloy specification satisfies one of the three groups of properties: basic, desired and application specific properties. The basic property includes rules that check if the program is well-formed, while the desired property includes rules that are beneficial for the program. The application-specific properties are made up of a number of basic and desired properties and optionally developer-specified properties.

3.7 Other

ESC/JAVA [22] is a compile-time program checker that uses a mix of annotations and extended static checking. The tool is able to detect a large number of automatically thrown exceptions. One of the problems with ESC/JAVA is that these will produce a warning even if they are caught and handled properly. It introduces the exsures pragma to handle exceptions which includes a boolean expression to indicate whether the exception should be thrown. This will be checked when the body of the annotated code is being analyzed. Listing 3 and the following explanation are taken from the ESC/Java user manual [22]:

```
1    exsures (T t) E ;opt
2    exsures (T) E ;opt
```

Listing 3: ESC/JAVA annotations

T is a subtype of java.lang.Throwable, t (if included) is a an identifier, and E is a boolean specification expression. The identifier t (if included) is in scope in E, where it has type T. The pragma makes E an exceptional postcondition of the routine the pragma modifies. That is, it specifies that E holds whenever the routine completes abruptly by throwing an exception t whose type is a subtype of T.
ESC/JAVA produces an AST while compiling the code and the annotations. This abstract syntax tree is then translated into guarded commands, which are then translated into verification conditions which are passed to a theorem prover.
Java Applet Correctness Kit (JACK) [6] is a tool for source and byte code verification that must be annotated with the Java Modeling Language. The tool uses an extended version of weakest precondition calculus, which includes various Java concepts such as exceptions. The two frameworks being used to handle the proof obligation that is being generated by the weakest precondition calculus are Simplify - an automatic prover, and Coq - an interactive prover.

3.8 Discussion

There are several ways to handle the exception flow when using helper variables. While the approach discussed in this thesis uses a global static variable in Java that can be accessed from anywhere in the program, the approach developed in „Interprocedural Exception Analysis for C++“[27] solves this problem by passing along a pointer to a local helper

variable that is supposed to represent the exception that is being thrown by a function that was called by another function. The authors discussed several reasons for this behavior, addressing the difference between exceptions in Java and exceptions in C++. One of the differences is the subtyping difference between Java and C++. Exceptions in Java all have a common ancestor class, the Exception class and only allow inheritance from one class while C++ allows a class to inherit from multiple classes.
The tool discussed in „Static analysis of exception handling in Ada“[31] can be considered to be similar to the approach discussed in this paper. The difference is where the exception should be handled. While the analysis itself is responsible for properly handling exceptions in the Ada tool, CPAchecker uses a helper variable and conditional statements to map the exceptional control flow on a CFA. The Ada implementation would also be possible in CPAchecker. While the Ada implementation would reduce the numbers of paths in a CFA, the advantage of the approach discussed in this thesis is that the analysis implementation does not become more complex. The Ada approach can be a disadvantage if the implementation of the analysis should be as simple as possible. The disadvantage of the approach discussed in this thesis is that a lot of paths will be added to the CFA. This adds to the already existing path explosion problem a lot of verification tools struggle with. Compared to TACO, CPAchecker has more freedom when it comes to implementing exceptional control flow. TACO also uses a helper variable in its process but is forced to do it because one of its intermediate languages do not offer exception handling, whereas CPAchecker could implement exception handling without the use of a helper variable. Both JBMC and the approach discussed in this thesis use a helper variable and conditional statements to translate exceptional control flow to non-exceptional control flow. In the case of JBMC, the translation results in a GO-TO program, while CPAchecker's translation results in a CFA.
The architectural exception flow analyses depend on a good understanding of the control flow in the program. Finding and defining all of the exceptional use cases can be quite difficult. Users often find ways to break a program that the software architect or developer didn't think of. Another problem with architectural exception flow analysis is that it can't inform the developer about implementation-specific problems, such as trying to access an array position outside of the arrays scope.
There are papers that compare different tools discussed in this chapter. In „Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites“[5] Java Pathfinder, ESC/JAVA and other tools are being compared. The authors used the Juliet Test Suite v1.2 as the dataset on which to test against. Java Pathfinder was quite precise, meaning that it could assign the correct problem when it found one but had problems finding the bugs in the first place. ESC/JAVA on the other hand found all of the known problems but performed rather poorly at assigning those bugs to the correct problem.
There is a comparison between JBMC, JayHorn and Java Pathfinder in „JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode“[17]. The programs used were from four different suites: a self-created suite, one made by Java Pathfinder, the MinePump suite and a recursive suite. JBMC performed quite well followed by Java Pathfinder. JayHorn, in comparison, was only able to correctly identify about half of the problems.

In „A comparison of bug finding tools for java“[30] 5 different bug finding tools are being compared including ESC/JAVA. Several categories are being discussed, including concurrency errors, null dereferences and array bounds errors. The programs used as a test suite were Apache Tomcat 5.0.19, JBoss 3.2.3, Art of Illusion 1.7, Azureus 2.0.7 and Megamek 0.29. Looking at the null dereferences ESC/JAVA does report many warnings in the different projects that were used to test these programs, up to a factor of 500 more warnings than the other tools tested. The time needed to analyze the different programs is also quite long, from at least one day up to fifteen days while the other programs often needed only a few hours or even minutes. This shows that ESC/JAVA doesn't really perform well without the annotations, which are either a big commitment to include for already written software or need to be considered from the beginning.
SV-COMP is a competition for software verification tools. It offers a Java category since 2019. The following results reference the report from 2023 [7]. The set of programs used can be found on their website[3] and includes test programs from different participants such as JayHorn, JBMC, JPF and Java Ranger. The participants that competed in the SV-COMP 2023 were GDart, Java-Ranger, JayHorn, JBMC, JDart, Coastal and SPF. The report includes two tables that show different results. The first one goes over all regular results and contains GDart, Java-Ranger, JBMC and MLB. The second table is a quantitative overview of all participants who want to participate in the competition without being ranked. The table contains the tools Coastal, JayHorn, JDart and SPF. JBMC and GDart performed by far the best when looking at the first table, both sitting at around 80% of the points. MLB managed to score around 60% of the points and Java-Ranger had close to 50% of the points. When looking at the second table JDart manages to get close to 50% of the points, while JayHorn and SPF both had around 25% of the points. Coastal was the only one in this category with a negative score. The scoring scheme can be found in the SV-COMP report „Competition on Software Verification and Witness Validation: SV-COMP 2023“[7].

[3] `https://sv-comp.sosy-lab.org/2023/benchmarks.php`

4 Adding Exception-based Control-Flow to CFAs

In this chapter, we will discuss an approach to building a CFA for a Java program with exceptions using the standard control flow. We will look at the different parts of the Java exception construct and how to translate them. Finally, an example of an abnormal execution will be translated into a CFA control flow using the methods discussed in this section. This concept is based on JayHorn's [4] implementation. The CFAs in this chapter will be simplified so as not to show any information that is unnecessary for the implementation. Listing 4 contains a program that is used as a running example in this chapter. The various Java exception statements are added to that program and translated into a CFA. In addition, we also show what the Java program would look like if source code with exceptional control flow would be translated to source code without exceptional control flow.

```
1  public class Main {
2      private static int i = 0;
3
4    private static void foo(){}
5    private static void bar(){}
6
7    public static void main(String[] args) {
8      foo();
9      bar();
10   }
11 }
```

Listing 4: Exceptional code without exception constructs

4.1 Global Helper Variable

The first step is to add a class to the program model that contains a `public static` helper variable of the type `Throwable`. The variable is initialized with a `null` value. This is shown in Listing 5. This variable represents an exception that is actively impacting the control flow.

```
1  public class Helper {
2      public static Throwable helperVariable = null;
3  }
```

Listing 5: Global helper - adding static helper variable

4.2 Throw Statement

A `throw` statement is represented in the CFA as an assignment of the `throw` statement expression to the helper variable. Execution of the method ends if the `throw` statement is not in a `try` block. This means that the path containing the `throw` statement goes directly to the end of the method or to a `finally` block if it was thrown in a `catch` block if a `finally` block exists in the same `try` statement as the `catch` block. If the `throw` statement is in a `try` block the normal exception handling mechanisms discussed Section 4.3 will apply. The

```
1  public class Main {
2      private static int i = 0;
3
4      private static void foo(){
5         throw new NullPointerException();
6      }
7      private static void bar(){}
8
9      public static void main(String[] args) {
10         foo();
11         bar();
12     }
13 }
```

Listing 6: Throw statement example

```
1  public class Main {
2      private static int i = 0;
3
4      private static void foo(){
5         Helper.helperVariable = new NullPointerException();
6      }
7      private static void bar(){}
8
9      public static void main(String[] args) {
10         foo();
11         bar();
12     }
13 }
```

Listing 7: Handling throw statement

result of translating the code in Listing 6 is the CFA in Figure 5 and the code in Listing 7.

4.3 Catching an Exception

There are two steps to implementing the exception handling. The first step adds a conditional statement that checks if the global helper variable is not `null`. These conditional edges are added after each edge that is a method call, a `throw` statement in a `try` block, or an abnormal execution. One path leads to the next statement in the program if the variable is `null`. The other path leads to the exception handling if the helper variable is not `null`. For each method that is used within another operation, e.g. `bar()` in `foo(bar())`, a temporary variable is declared with the return type of the method as a type and the method call is assigned to that variable. The operation uses the temporary variable as a value instead of the method call. After that step 1 can be applied to these methods as well. If there are no `catch` blocks in the source code the `helperVariable!=null` edge will go directly to the end of the method, or to the `finally` clause, if a `finally` clause exists if it is not nested. The nested cases will be discussed in Section 4.6

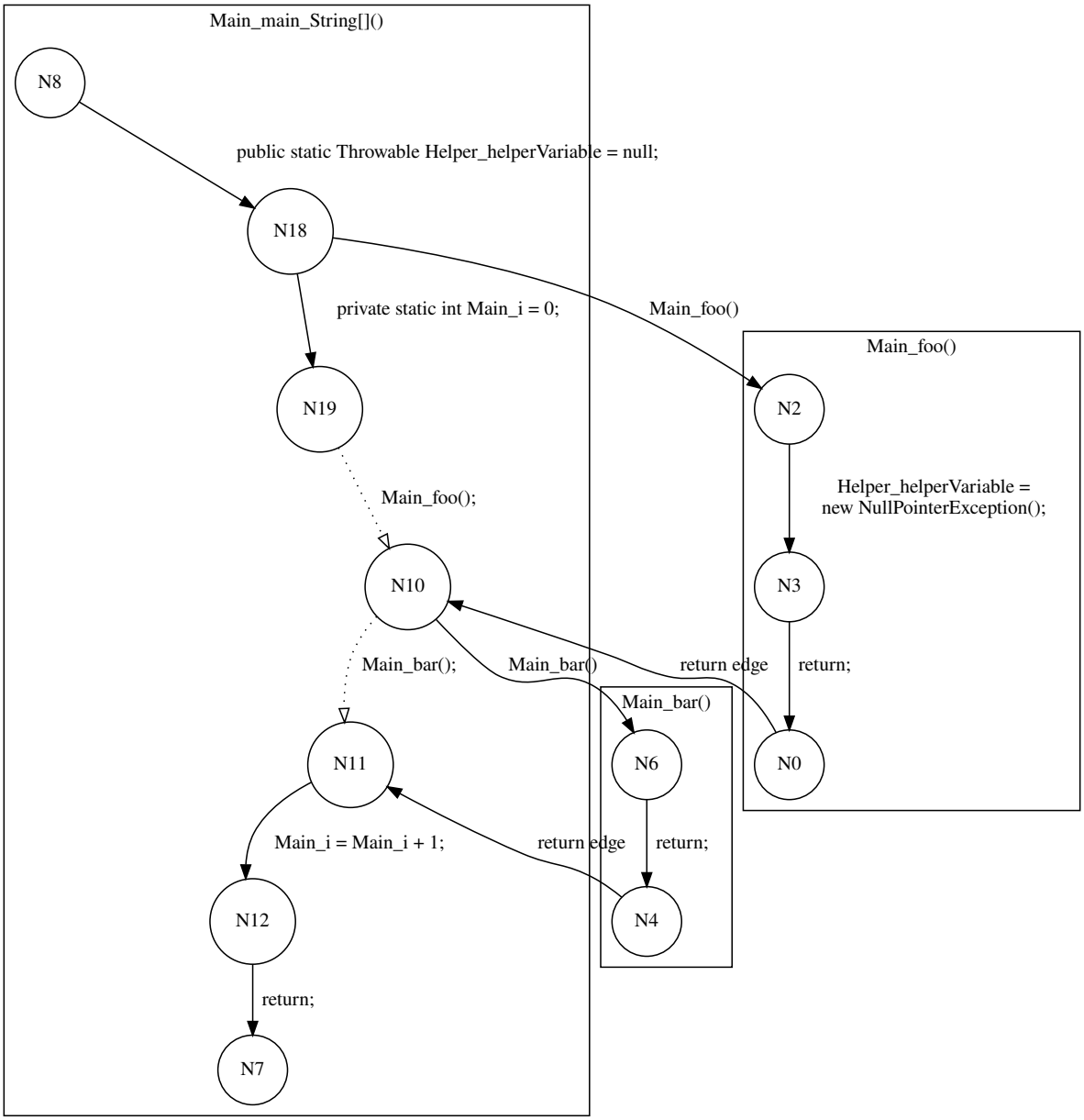

Figure 5: Adding throw statement handling to CFA

```
1  public class Main {
2     private static int i = 0;
3
4     private static void foo(){
5         throw new NullPointerException();
6     }
7     private static void bar(){}
8
9     public static void main(String[] args) {
10        try{
11          foo();
12          bar();
13        } catch(NullPointerException n){
14          n.printStackTrace();
15        } catch(RuntimeException r){
16          r.printStackTrace();
17        }
18    }
19 }
```

Listing 8: Example - try catch

In the second step, a conditional statement for each `catch` block will be added in the form of an `if-else if-else` construct that checks whether the helper variable is an instance of the class of the exception or a superclass of the class of the exception that is caught in the `catch` clause. The `catch`-block content will be the corresponding `if`-block content. At the beginning of each `if` block, a variable is declared with the type and the id of the CatchFormalParameter and initialized with the value of the helper variable. The helper variable must then be set to `null`. The `else` path leads either to a `finally` block, if available, or to the end of the method if the `try-catch` block does not include a `finally` block if it is not nested within another exception construct. The nested cases will be discussed in Section 4.6 The method continues normally if one of the `if-else if` blocks is executed. The way the two steps translate the example code in Listing 8 into a CFA is shown in Figure 6, and the source-to-source code translation is shown in Listing 9.

4.4 Finally Clause

The `finally` code block is always being executed regardless whether an exception is caught or not. An exception that is thrown in the `finally` block overrides an exception thrown before it or an exception that was not caught. Two different approaches to including a `finally` block in a CFA are discussed in this section.
The first is to add the content of the `finally` block at the end of all 3 paths before returning to the method body after the `try-catch` construct: helper variable is `null`, helper variable has a value but none of the `if-else if` statements catch it, and helper variable has a value and the value is caught by one of the `if-else if` cases. The paths continue after the `finally` block as discussed in the previous sections.
The second approach is to declare a boolean variable directly after the

```
1  public class Main {
2    private static int i = 0;
3
4    private static void foo(){
5      Helper.helperVariable = new NullPointerException();
6    }
7    private static void bar(){}
8
9    public static void main(String[] args) {
10     foo();
11     if( Helper.helperVariable != null){
12       if( Helper.helperVariable instanceof NullPointerException){
13         NullPointerException n = Helper.helperVariable;
14         Helper.helperVariable = null;
15         n.printStackTrace();
16       } else if( Helper.helperVariable instanceof RuntimeException) {
17         RuntimeException r = Helper.helperVariable;
18         Helper.helperVariable = null;
19         r.printStackTrace();
20       } else {
21         return;
22       }
23     } else {
24       bar();
25       if(Helper.helperVariable != null){
26         if( Helper.helperVariable instanceof NullPointerException){
27           NullPointerException n = Helper.helperVariable;
28           Helper.helperVariable = null;
29           n.printStackTrace();
30         } else if(Helper.helperVariable instanceof RuntimeException) {
31           RuntimeException r = Helper.helperVariable;
32           Helper.helperVariable = null;
33           r.printStackTrace();
34         } else {
35           return;
36         }
37       } else {}
38     }
39   }
40 }
```

Listing 9: Source-to-Source translation - catching exceptions

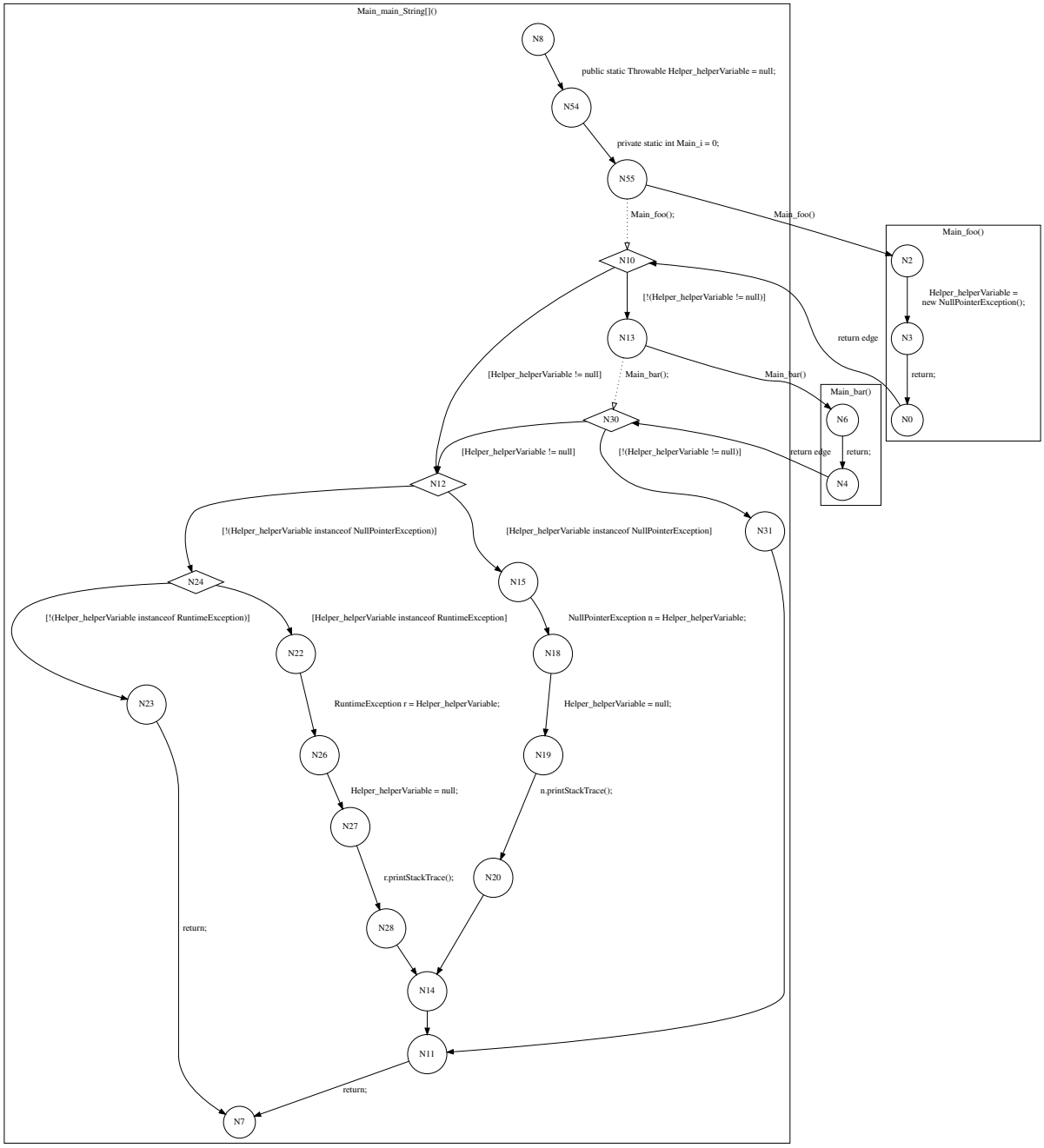

Figure 6: Adding exception catching control flow to CFA

```
public class Main {
  private static int i = 0;

  private static void foo(){
    throw new NullPointerException();
  }
  private static void bar(){}

  public static void main(String[] args) {
    try{
      foo();
      bar();
    } catch(NullPointerException n){
      n.printStackTrace();
    } catch(RuntimeException r){
      r.printStackTrace();
    } finally {
      i = 1;
    }
  }
}
```

Listing 10: Example including finally block

`helperVariable != null` check and initialize it with the value `false`. This variable keeps track of whether an exception was handled. At the end of each catch block that is represented by a conditional statement the variable gets assigned the value `true`, indicating that the exception was successfully handled. The boolean variable is also added and set to `true` at the end of to the normal execution path in the `try` block. The three paths that were mentioned in the first finally block approach lead to the same node. After this node, the `finally` block content will be added to the CFA. A conditional statement will be introduced after the finally block that checks whether the boolean variable is `true` or `false`. If it is `false` the path goes either to the end of the method or possibly to another exception handling construct when nested, as discussed in Section 4.6. It continues with the next statement of the method body if the boolean variable is `true`. While the first approach seems better when the `finally` block does not contain structures like loops that are hard to verify, the second approach should work better when many paths are introduced because the content is not added to all of the paths. The example code in Listing 10 is used to demonstrate both approaches. The first approach is shown in Figure 7 and Listing 11 and the second one can be seen in Figure 8 and Listing 12.

```
public class Main {
  private static int i = 0;

  private static void foo(){
    Helper.helperVariable = new RuntimeException();
  }
  private static void bar(){}

  public static void main(String[] args) {
    foo();
    if( Helper.helperVariable != null){
      if( Helper.helperVariable instanceof NullPointerException){
        NullPointerException n = Helper.helperVariable;
        Helper.helperVariable = null;
        n.printStackTrace();
        i = 1;
      } else if( Helper.helperVariable instanceof RuntimeException) {
        RuntimeException r = Helper.helperVariable;
        Helper.helperVariable = null;
        r.printStackTrace();
        i = 1;
      } else {
        i = 1;
        return;
      }
    } else {
      bar();
      if(Helper.helperVariable != null){
        if( Helper.helperVariable instanceof NullPointerException){
          NullPointerException n = Helper.helperVariable;
          Helper.helperVariable = null;
          n.printStackTrace();
          i = 1;
        } else if(Helper.helperVariable instanceof RuntimeException) {
          RuntimeException r = Helper.helperVariable;
          Helper.helperVariable = null;
          r.printStackTrace();
          i = 1;
        } else {
          i = 1;
          return;
        }
      } else {
        i = 1;
      }
    }
  }
}
```

Listing 11: Adding finally block to paths

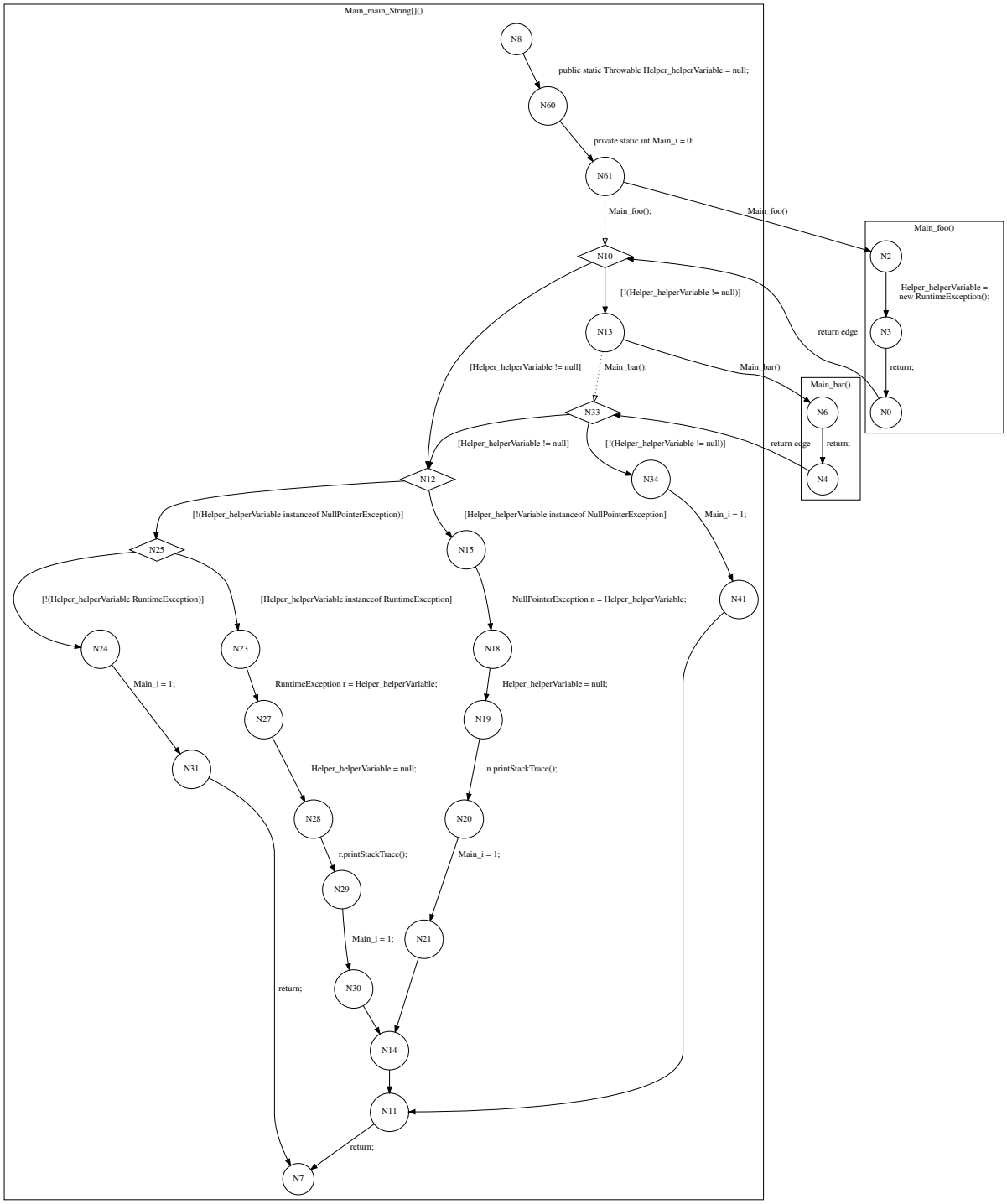

Figure 7: Finally - Adding Finally Block To All Eligible Paths

```
public class Main {
  private static int i = 0;

  private static void foo(){
    Helper.helperVariable = new RuntimeException();
  }
  private static void bar(){}

  public static void main(String[] args) {
    foo();
    if( Helper.helperVariable != null){
      boolean pathToEnd = false;
      if( Helper.helperVariable instanceof NullPointerException){
        NullPointerException n = Helper.helperVariable;
        Helper.helperVariable = null;
        n.printStackTrace();
        pathToEnd = true;
      } else if( Helper.helperVariable instanceof RuntimeException) {
        RuntimeException r = Helper.helperVariable;
        Helper.helperVariable = null;
        r.printStackTrace();
        pathToEnd = true;
      }
      i = 1;
      if(pathToEnd){}
      else {return;}
    } else {
      bar();
      if(Helper.helperVariable != null){
        boolean pathToEnd = false;
        if( Helper.helperVariable instanceof NullPointerException){
          NullPointerException n = Helper.helperVariable;
          Helper.helperVariable = null;
          n.printStackTrace();
          pathToEnd = true;
        } else if(Helper.helperVariable instanceof RuntimeException) {
          RuntimeException r = Helper.helperVariable;
          Helper.helperVariable = null;
          r.printStackTrace();
          pathToEnd = true;
        }
        i = 1;
        if(pathToEnd){}
        else{return;}
      } else {
        boolean pathToEnd = false;
        pathToEnd = true;
        i = 1;
        if(pathToEnd){}
        else{return;}
      }
    }
  }
}
```

Listing 12: Adding local variable to handle control flow after finally

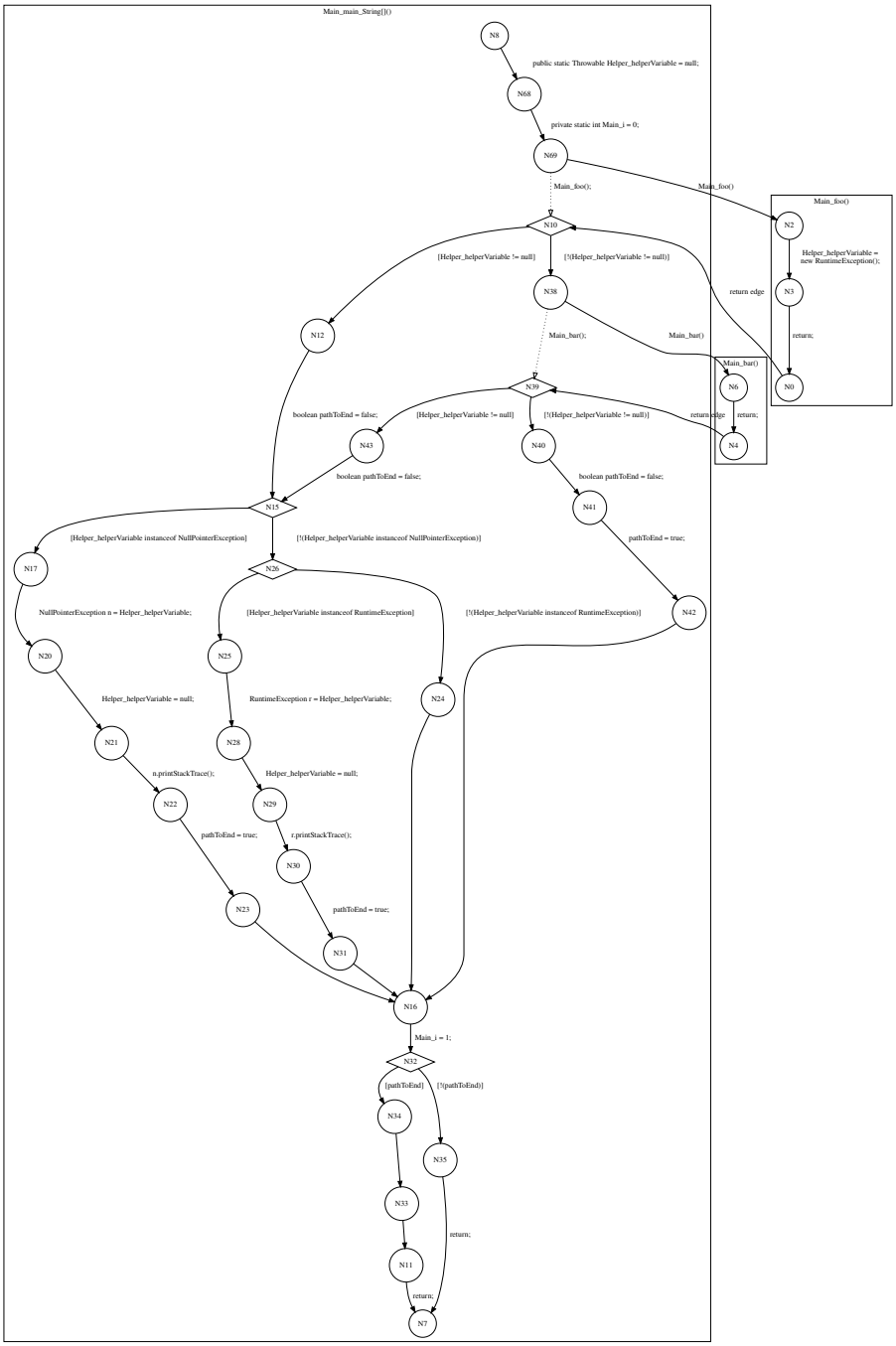

Figure 8: Adding local variable to handle control flow after finally in CFA

4.5 Throws Clause

The `throws` clause does not need to be handled since it is only an indication that a checked exception may occur, not a statement that an exception will occur. The global helper variable is already a stronger form of indication because it tells you exactly when an exception must be handled at any point in the program.

4.6 Nested Try Catch Finally

4.6.1 Nested in Try

Exception handling when a `try-catch(-finally)` construct is nested within a `try` block is similar to the normal exception handling. The difference is that the path representing that an exception occurred but was not caught in the `try-catch(-finally)` construct leads to the exception catching construct of the next outer `try` block. This can be seen in the CFA in Figure 9: it is the `[`(HelperVariable_helper instanceof ClassCastException)]! edge between node 14 and node 12. The rest of the code in the try block will be executed when no exception is thrown in the `try-catch(-finally)` construct or when the exception is properly handled.

4.6.2 Nested in Catch

When `a try-catch(-finally)` construct is nested within a `catch` block that is part of a `try` statement with multiple catches, if an exception is not caught by the `try-catch(-finally)` construct and the potential `finally` block already handled, the path leads to either a `finally` block that exists and is part of the same `try` statement as the `catch` clause, the end of the method or the next `catch` clause, if the catch block is also nested within a `try` statement. The execution of the rest of the code in the catch block continues when no exception is thrown in the `try-catch(-finally)` construct or when the exception is properly handled.

4.6.3 Nested in Finally

If an exception is thrown but not caught in a `try-catch(-finally)` construct that is nested within a `finally` block the execution gets interrupted and resumed at the next possible occasion. This could be either the end of the method, the next exception `catch` construct if the `finally` block was nested in a try statement or another `finally` block if the finally block was nested inside a catch which is followed by a `finally` statement. In the case that no exception is thrown in the `try-catch(-finally)` construct or the exception is properly handled, the execution of the rest of the finally block will be continued .

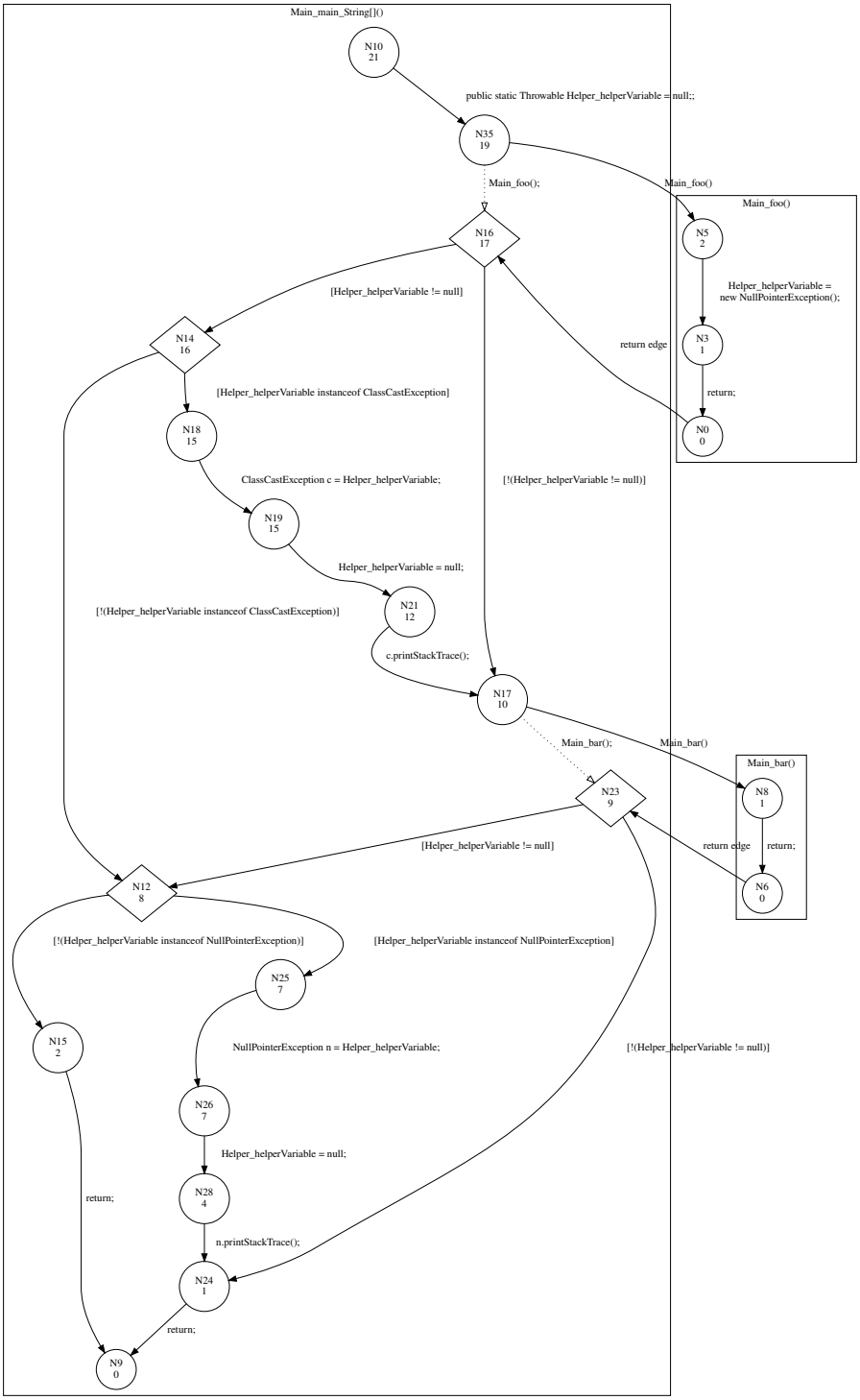

Figure 9: Nested in try block control flow

```
public class Main {
  public static void main(String[] args){
    int x = 0;
    int result = 5/x;
  }
}
```

Listing 13: Abnormal execution - division by zero

```
public class Main {
  public static void main(String[] args){
    int x = 0;
    int tempDivByZero;
    if(x == 0){
      Helper.helperVariable = new ArithmeticException();
      if(Helper.helperVariable != null){
        return;
      }
    } else {
      tempDivByZero = 5/x;
    }
    result = tempDivByZero;
  }
}
```

Listing 14: Division by zero control flow handled

4.7 Abnormal Execution - Division by Zero

There are abnormal executions that result in an exception that is not explicitly thrown by a developer using the `throw` keyword. Each of these cases requires its own way of handling. We will look at integer division by zero to demonstrate how these cases can be handled with the already discussed approaches. When encountering a division where the divisor is a variable, declare a temporary integer variable. Add conditional statements that check whether the variable in the divisor is zero. If it is not zero, assign the result of the division to the temporary variable and put the variable in the place where the division was before. Otherwise assign a new ArithmeticException instance to the helper variable and continue with the steps discussed in Section 4.3. Listing 13 is used as an example program. The result of the translation is shown in Figure 10 and Listing 14.

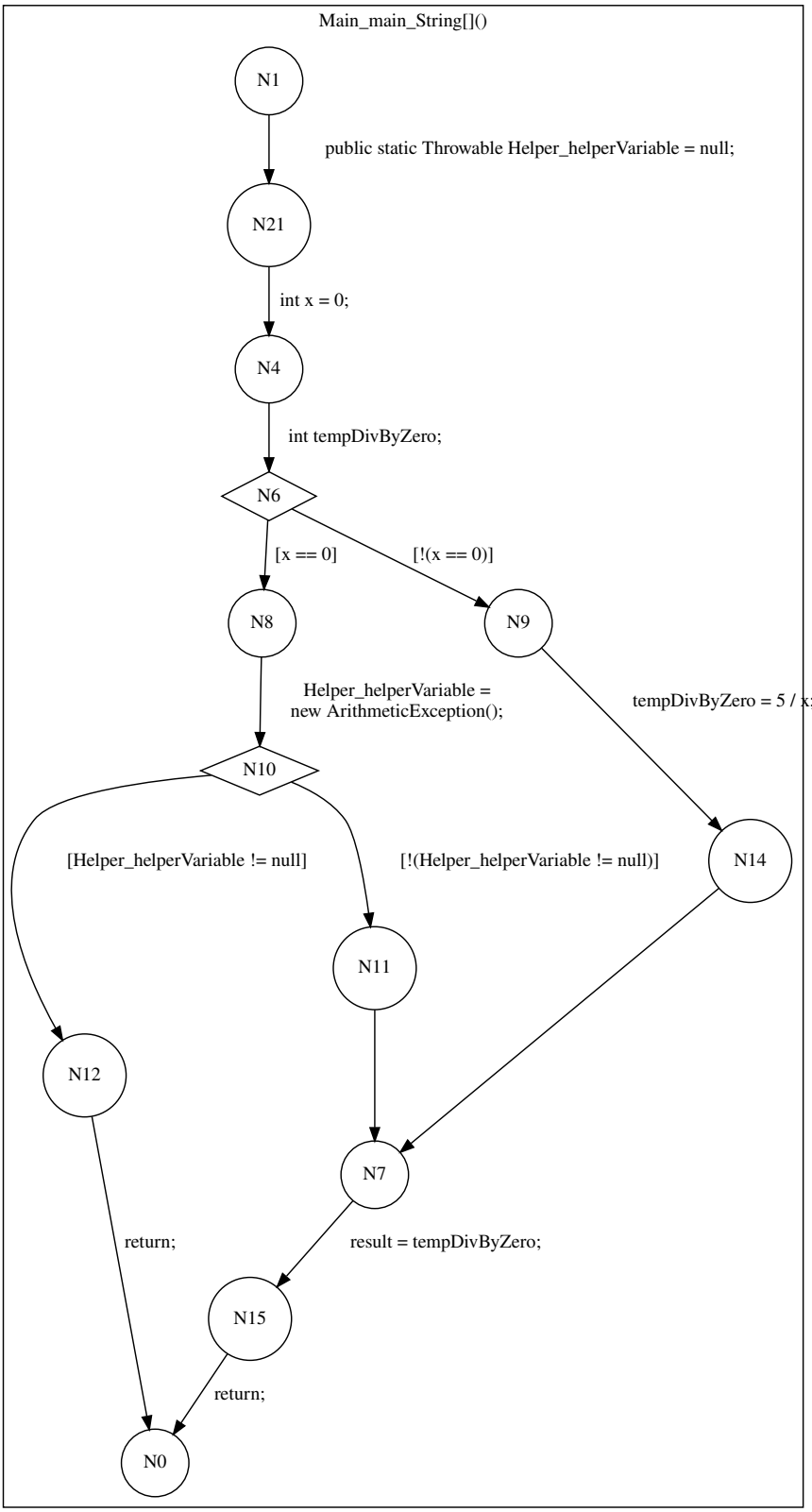

Figure 10: Division by zero control flow added to CFA

5 Evaluation

5.1 Setup

The following section answers two questions: how do the two branches javaExceptions[4] and javaExceptionsFinallyVariable[5] and CPAchecker trunk[6] compare to each other, and how do they compare to other state-of-the-art tools that participated in the SVCOMP 2023? A link to the exact version of the tools can be found in Table 7. All 3 CPAchecker programs used value analysis in conjunction with runtime-type analysis. The javaExceptions branch implemented the finally block in all eligible paths and the javaExceptionsFinallyVariable branch implemented the local finally handling variable approach, while none of the branches implemented abnormal execution scenarios. The evaluation was performed on the ws-cluster on the SoSyLab verifier cloud. Each machine in this cluster contains an Intel Core i7-6700 @3.400 GHz CPU with 8 cores and a frequency of 4 000 MHz with Turbo Boost disabled. The time limit is 120 s with a hard time limit of 150 s, and the memory limit is set to 3 000 MB. A Linux 5.15.0-88-generic operating system is installed on the machines. The benchmark consists of 608 programs: the current iteration of the SV-Benchmark JavaCategory[7] and 21 small programs that test different parts of user-handled exceptions[8]. All of these programs use the assert property to check for program correctness. Performance values are rounded to 3 significant digits.

5.2 Comparison between CPAchecker Implementations

In Table 1, you can see that CPAchecker trunk manages to solve 354 problems without running into a timeout or an error, while correctly solving 244 problems. Of the incorrectly solved problems, 3 were false positives and 107 were false negatives. It was able to solve some of the problems with exception constructs as mentioned in Section 2.9. There are 241 programs in the dataset that contain exception handling constructs. Trunk managed to correctly analyze 56 of these problems, incorrectly analyzed 38, and encountered an error, exception, or timeout in 147.
The javaExceptions branch manages to solve 355 problems while correctly solving 250 of them. It manages to correctly solve 18 problems that the CPAchecker trunk couldn't correctly solve. All of the 18 problems include some form of exception handling. There were 105 programs that were incorrectly identified. CPAchecker Trunk and javaExceptions share 92 of these. Of the other 13 problems that javaExceptions can't correctly identify, 10 are problems that contain abnormal execution scenarios. Of the 241 programs that contained exceptions, 62 were correctly analyzed, 33 were incorrectly analyzed, and 146 could not be analyzed due to some kind of error, exception, or timeout.
The javaExceptionsFinallyVariable branch only manages to solve 346 problems compared

[4] `https://svn.sosy-lab.org/software/cpachecker/branches/javaexceptions@45359`
[5] `https://svn.sosy-lab.org/software/cpachecker/branches/javaexceptionsfinallyvariable@45361`
[6] `https://svn.sosy-lab.org/software/cpachecker/trunk@45346`
[7] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/509aa68247e0050034f037ff56391fe87a8e4b0a/java`
[8] `https://svn.sosy-lab.org/software/cpachecker/branches/javaexceptions@45184`

CPAchecker	Overall	Correct			Incorrect		
		Results	True	False	Results	True	False
Trunk	354	244	82	162	110	3	107
javaExceptions	355	250	100	150	105	15	90
javaExceptionsFinallyVariable	346	244	100	144	102	15	87

Table 1: CPAchecker comparison

CPAchecker	Correctly Analyzed	Incorrectly Analyzed	Unknown
Trunk	56	38	147
javaExceptions	62	33	146
javaExceptionsFinallyVariable	56	30	155

Table 2: CPAchecker comparison: 241 programs containing exception handling

to javaExceptions. It can correctly analyze 244 of these problems. It solves 102 problems incorrectly, having the same problem with false positives as javaExceptions. When analyzing 241 programs that contained exceptions, 56 correct results were produced, 30 incorrect results were produced, and 155 could not be analyzed due to some kind of error, exception, or timeout in CPAchecker. The difference in solvable programs between javaExceptions and javaExceptionsFinallyVariable is due to a bug, not the approach.

The difference between the branches is negligible when comparing the performance of the 225 problems that are correctly analyzed by all 3 implementations as shown in Table 3 . This indicates that even if the exception constructs are being added to the CFA, they don't affect the performance of the tool in any meaningful way in this set of programs. This comparison can be made because the CPAchecker trunk manages to analyze some of the problems that involve exceptions, as discussed at the start of this section. From this point on, only trunk and javaExceptionsPath will be discussed, since there are no performance differences between javaExceptionsPath and javaExceptionsFinallyVariable in the current set of programs. Trunk will be called CPAchecker - Trunk and javaExceptionsPath will be called CPAchecker - Branch.

5.3 Comparison CPAchecker to State of the Art Software

In the following section CPAchecker and state-of-the-art Java verification tools get compared. The tools included are all tools that participated in the SV-Comp 2023: Coastal, GDart, Java-Ranger, JayHorn, JBMC, JDart, MLB and SPF. JBMC didn't perform as well

CPAchecker	cputime(s)	walltime(s)	memory(MB)	cpuenergy(J)
Trunk	1490	472	42400	12900
javaException	1510	480	43600	12900
javaExceptionsFinallyVariable	1490	478	43500	13000

Table 3: CPAchecker performance comparison: 225 programs where all CPAchecker versions get the correct result

Software	Correct		Incorrect		CPU time (s)	Walltime (s)	Memory (MB)	CPU energy (J)
	True	False	True	False				
CPAchecker - Trunk	82	162	3	107	21000	11600	255000	169000
CPAchecker - Branch	100	150	15	90	21000	11700	260000	170000
Coastal	196	210	97	0	12600	12500	62100	160000
GDart	192	309	0	0	18600	7870	372000	170000
Java-Ranger	197	275	0	0	10900	5610	217000	103000
JayHorn	122	94	1	0	31200	15000	557000	303000
JBMC	109	140	0	0	1130	530	32200	9800
JDart	209	328	0	0	10400	8020	271000	93900
MLB	180	272	0	0	8800	6600	94200	104000
SPF	122	92	0	0	1490	593	35800	13400

Table 4: Software comparison complete dataset

as it did in the SV-Comp 2023 due to an error occurring, but is still discussed in the performance comparison.

The results in Table 4 show that JDart, GDart, Java-Ranger and MLB all perform well, being able to validate at least 450 of the 601 given problems and not producing any incorrect results. Coastal comes close to the mentioned tools with 406 correctly analyzed problems, but it produces 97 false positives. The remaining tools, including CPAchecker-branch and trunk, JayHorn and SPF, are only able to correctly identify between 210 and 250 problems. CPAchecker is performing by far the worst of these tools, producing a large number of false positives and false negatives.

Table 5 shows that there is quite large difference when looking at the performance of the tools. The set of problem considered are the 63 problems that all of the tools can correctly identify. The quantile plots discussed in this section include all problems that the individual tools could solve correctly.

SPF, JBMC, Java-Ranger, and Coastal are the best performers when it comes to CPU time, all staying under 150 s. MLB and JDart both take a little over 200 s. CPAchecker managed to correctly analyze the problems in about 6 min. GDart and JayHorn are the only programs that take close to or more than 10 min to correctly analyze all 63 problems. Looking at the quantile plot in Figure 11, both CPAchecker implementations, JBMC and SPF all finish their analysis quite fast but don't have many problems that they can correctly analyze. This seems to be due to the fact that they weren't able to solve the more complex problems. JayHorn does produces just as many results correct results but takes much more time.

There isn't that much difference between the tools when looking at the walltime. JBMC, SPF, Java-Ranger all need less than 60 s, JBMC only needing 33.7 s. MLB as well as Coastal and both CPAchecker implementations do solve the problems in close to 100 s to 120 s walltime. GDart and JDart are close with about 2.5 min, while JayHorn takes by far the longest with about 4.5 min. The quantile plot in Figure 12 shows a similar result to the previos quantile plot in Figure 11.

JBMC, Coastal, SPF and Java-Ranger have the lowest memory consumption of all the tools at around 4 000 MB or less, closely followed by MLB at 6 750 MB. Both CPAchecker implementations use almost twice as much memory as MLB. JDart and JayHorn have a similar memory consumption, requiring 15 600 MB and 17 800 MB, respectively. GDart

Software	CPU time (s)	Walltime (s)	Memory (MB)	CPU energy (J)
CPAchecker - Trunk	368	120	11100	3240
CPAchecker - Branch	370	122	11100	3240
Coastal	142	109	3110	1590
GDart	575	170	29100	4740
Java-Ranger	139	58.8	4040	1280
JayHorn	778	264	17800	6780
JBMC	67.7	33.7	2840	609
JDart	227	150	15600	2370
MLB	194	92.5	6750	1810
SPF	119	52.4	3270	1090

Table 5: Tool performance comparison: 63 programs where all tools get the correct result

has by far the largest memory consumption at almost 30 000 MB. Most of the tools use around 1 000 MB and 2 000 MB when comparing the tools in the quantile plot in Figure 13. SPF and JBMC appear to be the most memory efficient programs overall, staying under 100 MB per program for the most part. The rest of the tools do seem to have some problems that consume quite a lot of memory.
JBMC, SPF, Java-Ranger, Coastal and MLB are the most energy efficient tools. They all consume less than 2 000 J, with JBMC consuming less than 700 J. JDart and CPAchecker are in the middle of the pack when it comes to energy consumption. GDart and JayHorn perform the worst compared to the rest of the tools, consuming more than 4 500 J. The same can be seen in Figure 14.
When considering only those programs that contain at least one of the following exception keywords: „try“„„catch“„„finally“or „throw“, GDart, JDart and Java-Ranger perform the best. As shown in Table 6 they are able to correctly identify between 188 and 199 of the 241 problems and don't produce any false positive or false negative results. MLB also performs quite well, correctly analyzing 150 of the 241 problems. Coastal still manages to produce 106 correct results, but also has 95 incorrect results too. CPAchecker, JayHorn and SPF perform the worst, only being able to correctly analyze between 40 and 70 of the programs. CPAchecker does also produce 33 incorrect results when looking at CPAchecker-Branch and 38 when looking at CPAchecker-Trunk.

5.4 CPAchecker Possible Improvements

There are quite a few programs that CPAchecker either can't analyze correctly or can't handle at all. Programs with exceptions can't currently be handled by CPAchecker - Trunk. This includes abnormal execution, e.g. division by zero, and a way to include potential exceptions from other libraries, which can't be handled by the CPAchecker - Branch either. One approach that could be implemented is to make the helper variable non-deterministic after a library method is encountered and check both paths. CPAchecker does currently not support anonymous classes, which leads to an assertion error when analyzing the securibench problems. The replace5_eqchk problems from the java-ranger-regression folder cause an exception because the increment operation on an array index position is not properly handled. There is currently a bug in the ErrorPathShrinker class

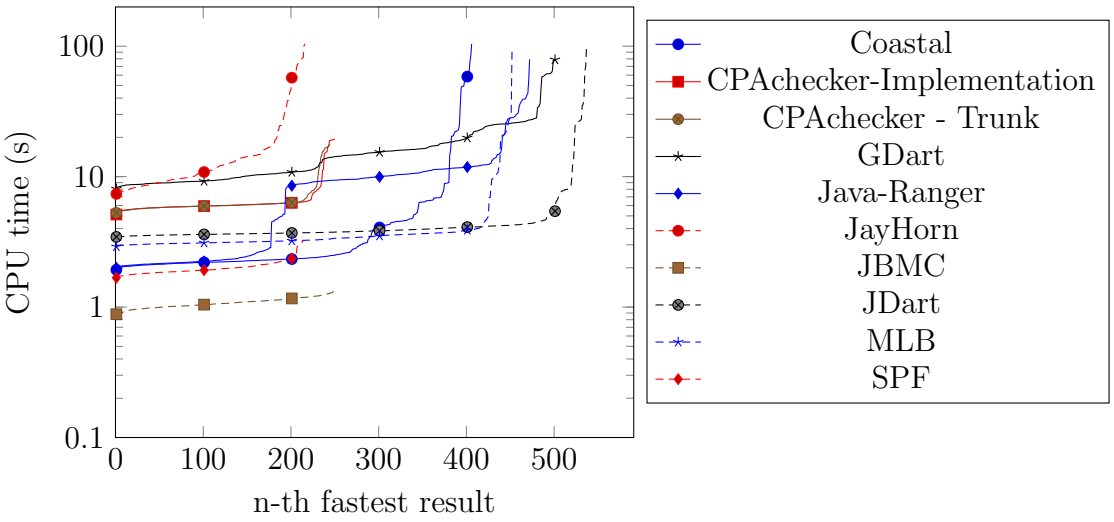

Figure 11: Quantile plot: CPU time of all correct results

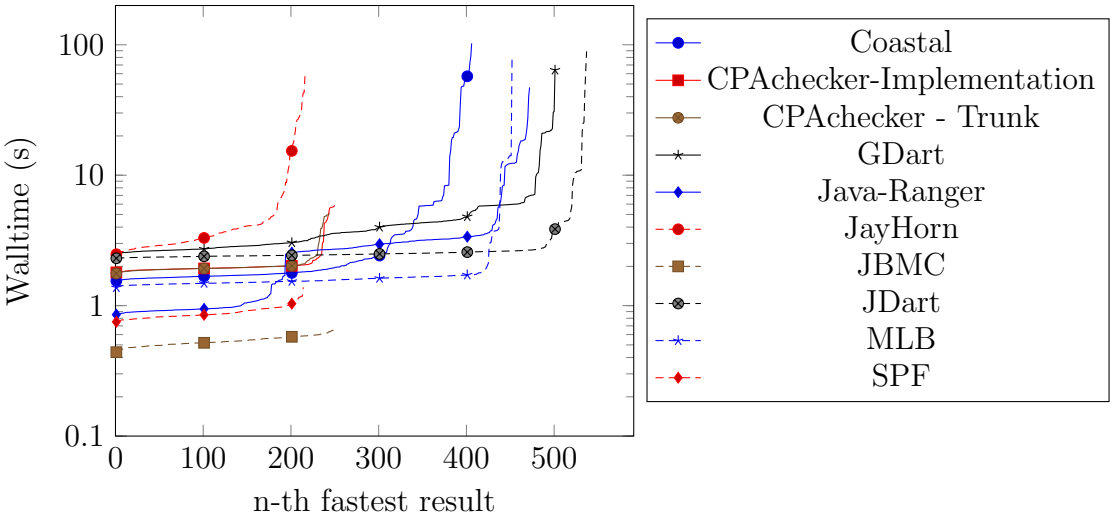

Figure 12: Quantile plot: walltime of all correct results

Software	Correctly Analyzed	Incorrectly Analyzed	Unknown
CPAchecker - Trunk	56	38	147
CPAchecker - Branch	62	33	146
Coastal	106	95	40
GDart	188	0	53
Java-Ranger	181	0	60
JayHorn	42	0	199
JBMC	65	0	176
JDart	199	0	42
MLB	150	0	91
SPF	50	0	191

Table 6: Software Comparison: 241 programs containing exception constructs

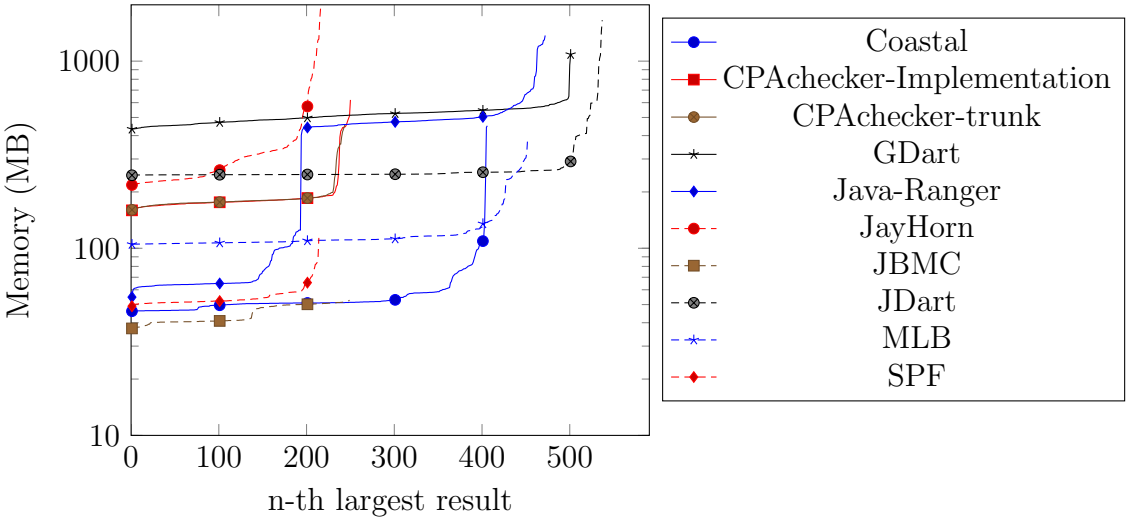

Figure 13: Quantile plot: memory consumption of all correct results

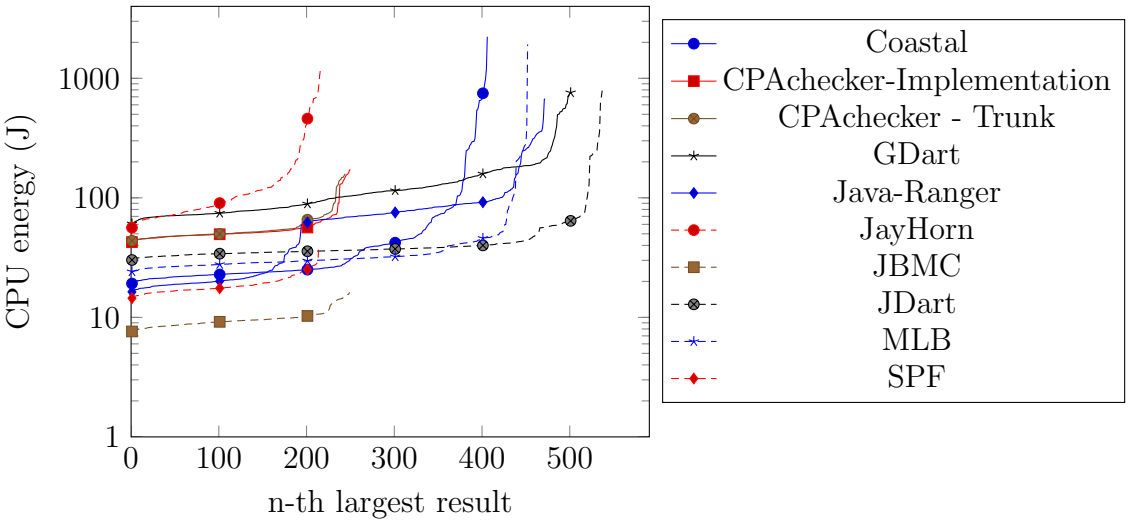

Figure 14: Quantile plot: CPU energy consumption of all correct results

in CPAchecker that can't handle objects that don't need to be declared. An example for such a scenario would be jbmc-regression/basic1. There is another bug in the CFA creation when including a method with a return type, where one of the paths doesn't return a value but throws an exception. Another problem is that the main method parameter does not get its own local variable, which leads to problems in situations using the `instanceof` operator with the string array parameter of the main method. Explicit value analysis and runtime type analysis cannot properly handle random values; all paths that rely on that value are explored, leading to incorrect false results if one of those paths leads to an assert false statement, even though that statement would never be reachable. This could be improved by changing the other analyses to handle Java programs.

6 Conclusion and Future Work

In this paper we introduced an exception handling concept based on introducing a global variable to keep track of an exception that is actively affecting the program, and conditional statements to check whether the exception can be handled. We also analyzed the current state of exception handling in verification and verification adjacent areas and discussed the different approaches.
Afterwards we discussed how the new CPAchecker implementations compare to each other, to the current state, and to 8 state-of-the-art Java verification tools. There is no difference in performance between the two finally implementations when being compared with the used dataset. We proved that the new exception handling system improves the Java verification capability of CPAchecker. The current state of CPAchecker and the new implementations that include the exception handling are currently not able to compare to the other tools and still need some improvement. To be able to handle all types of exceptions, abnormal execution needs to be implemented in the future. Another problem is that CPAchecker is not able to handle exceptions that are thrown from included libraries. In Section 5.4 we discussed non-exception issues that CPAchecker needs to improve in order to be comparable to the other tools. An interesting topic to research would be a performance comparison between the approach discussed about in this thesis and an implementation of the exception control flow in the analysis.

A Appendix

Tool	Link
Coastal	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/coastal.zip
GDart	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/gdart.zip
Java-Ranger	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/java-ranger.zip
JayHorn	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/jayhorn.zip
JBMC	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/jbmc.zip
JDart	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/jdart.zip
MLB	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/mlb.zip
SPF	https://gitlab.com/sosy-lab/sv-comp/archives-2023/raw/svcomp23/2023/spf.zip

Table 7: Link to version of tools used in evaluation

References

[1] Jayhorn github readme. `https://github.com/jayhorn/jayhorn#readme`. [Online; last accessed 29-11-2023].

[2] Oracle specification: Chaper 11. exceptions. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-11.html`. [Online; last accessed 29-11-2023].

[3] Oracle specification: Programming with assertions. `https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`. [Online; last accessed 29-11-2023].

[4] Alexdi. Getting rid of implicit control flow. `http://jayhorn.github.io/jayhorn/jekyll/2016/08/02/implicit-control-flow/`, August 2016. [Online; last accessed 29-11-2023].

[5] R. Amankwah, J. Chen, H. Song, and P. K. Kudjo. Bug detection in java code: An extensive evaluation of static analysis tools using juliet test suites. *Software: Practice and Experience*, 53(5):1125–1143, 2023.

[6] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[7] D. Beyer. Competition on software verification and witness validation: Sv-comp 2023. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522, Cham, 2023. Springer Nature Switzerland.

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, LNCS 4590, pages 504–518. Springer-Verlag, Heidelberg, 2007.

[9] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification, 2009.

[10] D. Beyer and S. Löwe. Explicit-value analysis based on CEGAR and interpolation. Technical Report MIP-1205, Department of Computer Science and Mathematics (FIM), University of Passau (PA), December 2012.

[11] P. Bjesse. What is formal verification? *SIGDA Newsl.*, 35(24):1–es, dec 2005.

[12] P. H. Brito, R. De Lemos, C. M. Rubira, and E. Martins. Architecting fault tolerance with exception handling: Verification and validation. *Journal of Computer Science and Technology*, 24(2):212–237, 2009.

[13] N. Cacho, E. Barbosa, J. Araujo, F. Pranto Filho, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, and I. Garcia. How does exception handling behavior evolve? an exploratory study in java and c# applications. 09 2014.

[14] F. Castor Filho, P. H. da S. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79(10):1397–1418, 2006. Architecting Dependable Systems.

[15] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for java. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, SAC '01, page 620–625, New York, NY, USA, 2001. Association for Computing Machinery.

[16] M. Chicote, D. Ciolek, and J. P. Galeotti. Practical jfsl verification using taco. *Software: Practice and Experience*, 44(3):317–334, 2014.

[17] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik. Jbmc: A bounded model checking tool for verifying java bytecode. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 183–190, Cham, 2018. Springer International Publishing.

[18] L. Cordeiro, D. Kroening, and P. Schrammel. Jbmc: Bounded model checking for java bytecode. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 219–223, Cham, 2019. Springer International Publishing.

[19] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975.

[20] M. J. Frade and J. S. Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252–277, 2011.

[21] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2:366–381, 2000.

[22] K. Rustan and M. Leino, Greg Nelson and James B. Saxe. Esc/java user's manual. `https://www.kindsoftware.com/products/opensource/escjava2/esctools/docs/escjava-usersmanual#ESC`, October 2012. [Online; last accessed 29-11-2023].

[23] T. Kahsai, P. Rümmer, and M. Schäf. Jayhorn: A java model checker. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 214–218, Cham, 2019. Springer International Publishing.

[24] I. Kádár, P. Hegedűs, and F. Rudolf. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, Jan. 2014.

[25] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. Jdart: A dynamic symbolic analysis framework. In M. Chechik and J.-F. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[26] M. Ouimet and K. Lundqvist. Formal software verification: Model checking and theorem proving. *Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA*, 2007.

[27] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta. Interprocedural exception analysis for c++. In M. Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 583–608, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[28] C. S. Pundefinedsundefinedreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 15–26, New York, NY, USA, 2008. Association for Computing Machinery.

[29] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, apr 2003.

[30] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *15th International symposium on software reliability engineering*, pages 245–256. IEEE, 2004.

[31] C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in ada. *Software: Practice and Experience*, 23(10):1157–1174, 1993.