Semantic Refactorings

Pascal Kesseli
St Cross College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Hillary 2017

Abstract

Refactorings are structured changes to existing software that leave its externally observable behaviour unchanged. The intent is to improve readability, performance or other non-behavioural properties of a program. Agile software engineering processes stress the importance of refactoring to keep program code extensible and maintainable. Despite their apparent benefits, manual refactorings are time-consuming and prone to introducing unintended side effects. Research efforts seek to support and automate refactoring tasks to overcome these limitations.

Current research in automatic refactoring, as well as state-of-the-art automated refactoring tools, frequently rely on *syntax*-driven approaches. They focus on transformations which can be safely performed using only syntactic information about a program or act overly conservative when knowledge about program semantics is required. In this thesis we explore *semantics*-driven refactoring, which enables much more sophisticated refactoring schemata. Our *semantics*-driven refactorings rely on formal verification algorithms to reason over a program's behaviour, and we conjecture they are more precise and can handle more complex code scenarios than *syntax*-driven ones.

For this purpose, we present and implement a program synthesis algorithm based on the CEGIS paradigm and demonstrate that it can be applied to a diverse set of applications. Our synthesiser relies on the bounded model checker CBMC [22] as an oracle and is based on an earlier research prototype called Kalashnikov [29]. We further define our Java Stream Theory (JST) which allows us to reason over a set of interesting semantic refactorings. Both solutions are combined into an automated semantic refactoring decision procedure, reasoning over program behaviours, and searching the space of possible refactorings using program synthesis. We provide experimental evidence to support our conjecture that *semantics*-driven refactorings exceed *syntax*-driven approaches in precision and scope.
Keywords: refactoring, formal verification, program synthesis

Acknowledgements

I would like to thank my supervisor, Professor Daniel Kroening, for giving me the chance to pursue research at this prestigious institution, and for his continued support and encouragement throughout my degree. I also thank my co-supervisor, Dr Cristina David, from whom I learned a great deal over the course of my DPhil, and who was always ready with encouraging words in times of need. I further thank all my colleagues at the Systems Verification Group, with whom I had the pleasure of sharing many research projects, each of which was unique, challenging and rewarding in its own way.

Finally, I thank Ms Xiao Fei Song for her unwavering support over the course of the three years of my degree, for every helping hand, every endearing encouragement, and every word of advice she provided.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Agile software engineering describes a category of software development processes based on the evolutionary creation of software. Such approaches focus on cooperation with stakeholder requirements combined with flexibility towards changes in the program source code at any time during the development process. This approach stands in direct contrast to traditional, milestone-based project processes for which late changes in requirements or program behaviour represent exceptional events. Agile software engineering embraces change at any stage of the project, which may be driven by a variety of reasons, such as customer requests or technological limitations. The popularity of these agile development approaches grew substantially in the past decade and gave rise to software development frameworks, such as SCRUM, Extreme Programming (XP), and Rapid Application Development (RAD) [34, Ch. 2]. Software projects developed in this fashion begin with early drafts and prototypes of desired features, which are then continually rewritten, expanded, and improved as the specification for the software evolves.

To implement agile engineering processes, developers must continually adapt and improve existing source code in a structured manner. In certain cases, this is done to change the semantics of the program by introducing new features or changing existing behaviour. Frequently, however, program code needs to be improved without changing its externally observable behaviour. Such changes are called refactorings and are introduced to improve the structure and readability of a program's source code while maintaining its current semantics [61, Ch. 3].

Refactoring is the process of adapting and restructuring existing source code without changing its externally observable behaviour. The intent of refactoring is

```
void intersect(List lhs, List rhs)
{
  Iterator it = lhs.iterator();
  while (it.hasNext()) {
    Object e = it.next();
    if (!rhs.contains(e))
      it.remove();
  }
}
```

(a) Redundant algorithm in Java.

```
void intersect(List lhs, List rhs)
{
  lhs.retainAll(rhs);
}
```

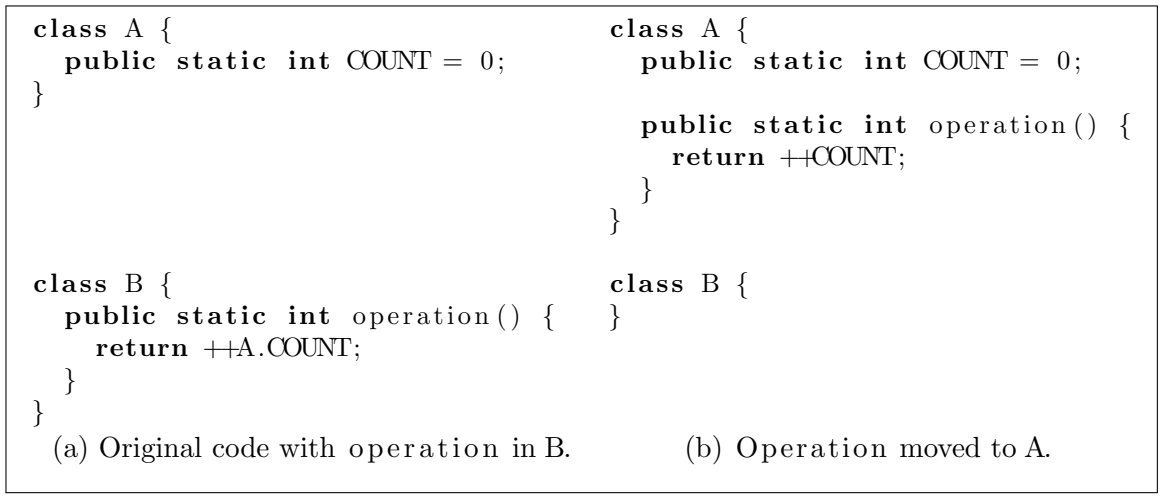
(b) Refactored utility function call.

Figure 1.1: Replace redundant algorithm (a) by existing utility function in Java (b).

not to introduce new features or repair faults, but to improve the non-functional aspects of the software. As an example, refactorings may eliminate redundancy, simplify an existing algorithm or improve a program's structure. Other non-functional improvements to software also include performance optimisations, such as reducing an algorithm's CPU and memory consumption.

Refactoring Example: Fig. 1.1 provides a simple example of a typical source code refactoring in the Java programming language illustrating an intersection algorithm for two input lists in Fig. 1.1a. This code segment is redundant since the Java runtime library provides the `retainAll` operation for its collection types providing the same semantics. The source code is considered refactored in Fig. 1.1b with the use of this function instead.

While refactorings are an accepted and important part of agile software engineering processes, they impose a non-trivial workload on software developers. Spotting instances such as the one presented in Fig. 1.1a, requires detailed code review sessions as incorrectly applied refactorings can introduce unexpected bugs. Therefore it is desirable to automate and support the refactoring process with a formal verification that the refactored code satisfies the desired properties. Based on the related work in the area (cf. Sec. 3.4), we assess this effort is accomplished only for one specific sub-category of refactorings and remains largely unsolved for the remainder. We explain this perceived dichotomy in refactoring automation in the following section.



```
class A {
  public static int COUNT = 0;
}

class B {
  public static int operation() {
    return ++A.COUNT;
  }
}
```

(a) Original code with `operation` in B.

```
class A {
  public static int COUNT = 0;

  public static int operation() {
    return ++COUNT;
  }
}

class B {
}
```

(b) Operation moved to A.

Figure 1.2: Move Method refactoring from class B (a) to A (b).

1.2 State of the art

We divide the set of refactorings defined by Fowler et al. into the two categories of syntax- and semantics-driven. The former are code changes which can be performed without having to understand or provide a model for the changed program's behaviour. These are usually conservative and purely syntactical changes, such as the *Move Method* refactoring [34, Ch. 7] illustrated in Fig. 1.2. The method `operation` is moved from class B in Fig. 1.2a to class A in Fig. 1.2b, with the intent of simplifying the program syntactically, since `operation` is solely dependent on the field COUNT in class A.

To perform this refactoring, no actual understanding of the semantics implemented in `operation`, i.e., the fact that it increments COUNT, is required. A syntactic analysis of the method, such as a dependency analysis on the program's abstract syntax tree, is sufficient to perform it safely. We refer to this category of refactorings as "syntax-driven". By contrast, the preconditions for the redundancy removal in Fig. 1.1a are much harder to establish. To prove redundancy and replace the code with the existing method, a decision procedure needs to confirm that both code snippets are equivalent under any possible input. The refactoring decision, in this case, is entirely dependent on the semantics of the program in question, which is why this category of refactorings is referred to as "semantics-driven".

The subject of syntax-driven refactorings has been extensively researched, and industrial software exists to support developers in this task. A comprehensive list of related publications is included in Sec. 3.4. The research presented in this thesis

is exclusively focused on semantics-driven refactorings, where a model of program semantics is a key requirement to perform the necessary code changes soundly.

1.3 Research goal

The goal of this thesis is to develop an approach for automating semantics-driven refactorings. We first present two specific semantics-driven refactorings followed by an outline of the methodology used for developing an automated reasoning engine to apply the refactoring to generic source code.

Remove unnecessary reflection: The Java Reflection API is a language feature provided by the Java programming language allowing programmers to access an in-program representation of the program's own source code. This is useful to examine and adapt the behaviour of a Java program at runtime [85]. We provide a example code of a reflection feature in Fig. 1.3 that executes all methods in the class `ReflectionTest` prefixed with `test`.

```
class Test {
  static void testX() {
    // ...
  }
  static void testY() {
    // ...
  }
  static void closeX() {
    // ...
  }

  static void reflectionInvoke() throws Exception {
    for(Method method : Test.class.getDeclaredMethods()) {
      if (method.getName().startsWith("test"))
        method.invoke(null);
    }
  }
```

Figure 1.3: Reflection example invoking all methods prefixed with "test".

Reflection is a flexible and powerful feature used to express semantics over the program structure. However, the use of reflection comes at the cost of an additional level of indirection, which negatively impacts performance and code complexity. Its use is discouraged if the same semantics can be accomplished using regular Java language features. In the context of automatically generated or translated Java source

code, unnecessary use of the Reflection API is an undesirable side effect. An example of unnecessary Reflection use is illustrated in Fig. 1.4a as well as an appropriate refactoring for this situation in Fig. 1.4b.

```
class Peer {
  static void op() {
    Class<?> cls = Test.class;
    Method method =
      cls.getDeclaredMethod("testX");
    method.invoke(null);
  }
}
```

(a) Static method call in Reflection.

```
class Peer {
  static void op() {
    Test.testX();
  }
}
```

(b) Explicit method call without indirection.

Figure 1.4: Unnecessary reflection (a) and its removal (b).

Replace Loop by Java 8 Stream query: Domain-specific languages (DSL) are programming languages or sub-languages within a programming language intended to facilitate programming within a specific domain. The Java Stream framework is a DSL introduced to Java in version 8. Its purpose is to facilitate the extraction of information from Java collection types. This allows programmers to implement algorithms which retrieve data from Java collections in an explicit query language, hence conveying the intent of the program more clearly. Fig. 1.5 illustrates this benefit by implementing a common filtering algorithm, using an explicit loop in Fig. 1.5a, and its equivalent Java 8 Stream query in Fig. 1.5b.

```
List<String> l = getData();
Iterator<String> it = l.iterator();
while (it.hasNext())
{
  String e = it.next();
  if (e.length() > 10)
    it.remove();
}
```

(a) Original code with loop.

```
List<String> l = getData();
l = l.stream()
    .filter(e -> e.length() <= 10)
    .collect(Collectors.toList());
```

(b) Equivalent Java 8 Stream query.

Figure 1.5: Explicit filter implementation (a) and Java 8 Stream filtering (b).

Introducing a domain-specific language to transform commonly used semantics into explicit instructions of an interpreter language is a special case of the “Replace

Implicit Language with Interpreter" refactoring described by Kerievsky in [61, p.269]. It increases code readability by explicitly naming the performed actions using instructions in the introduced DSL. For example, instead of using a generic `while-if` construct as in Fig. 1.5a, the refactored version in Fig. 1.5b invokes the Java 8 `filter` operation with an appropriate property. This conveys the intended semantics that elements in the list are to be removed according to a provided property.

Key ideas developed in this thesis: The two refactorings require semantic knowledge of the input source code to be applied soundly. In the case of "Remove unnecessary reflection", it needs to be first established that the reflection is unnecessary and that the invoked function is always the same under every possible execution. For the "Replace Loop by Java 8 Stream query", we need to determine if there exists a query in the Java 8 Stream domain-specific language which is equivalent to the current code. We hypothesize is that these semantic preconditions to these refactorings can be established by using the bounded model checker CBMC [22] (cf. Sec. 2.1).

While the first problem is expressible in CBMC, the refactoring to Java 8 streams requires us to prove that a stream query is equivalent to the existing source code and identify an equivalent stream query out of a set of possible queries in the allowed Java 8 Stream grammar. Other than the reflection refactoring, the stream refactoring represents a program synthesis problem we solve by presenting a refactoring synthesis engine using CBMC as an oracle.

Methodology:

1. Establish that CBMC is suitable to verify properties relevant to semantic refactoring preconditions. We demonstrate this by implementing the reflection refactoring as an exemplar in Chapter 4. We provide a selection of JNI-enabled benchmarks to corroborate our hypothesis that CBMC is sufficient to establish preconditions relevant to refactorings.

2. Develop a program synthesis engine with CBMC as an oracle which is scalable enough to handle programs comparable to our Java 8 stream refactoring problem. This step is presented in Chapter 5. We evaluate our synthesis engine using relevant benchmarks from renowned software verification competition sources. This evaluation is presented as evidence that our engine is of practical use for synthesis problems similar in complexity to our refactoring synthesis use case.

3. Formalise the refactoring constraints for the Java 8 stream refactoring problem and combine them with the previously implemented program synthesis engine to establish a refactoring synthesis engine. We present this step in Chapter 6. Randomly selected Java benchmarks from open source repositories are used to compare our semantics-driven refactoring engine against traditional syntax-driven refactoring tools.

1.4 Outline

This thesis consists of contributions in the field of automated, *semantics*-driven refactorings. The presented research investigates the necessary techniques to implement a fully automated program refactoring algorithm. Chapter 1 introduces the agile software engineering literature related to program refactorings with explanations of the methodology as well as current research efforts and limitations. A brief description of preliminary techniques and frameworks follows in Chapter 2.

In Chapter 4, we present the first exemplar of an automated, *semantics-driven* code transformation engine. The presented algorithm is an automated source code translation and minimisation engine that translates native JNI/C code fragments to the Java programming language. We further illustrate cases where semantic properties need to be established using a bounded model checker to minimise the translated source code safely. The chapter concludes with the implementation of our proposed algorithm in our tool *Jitsune*, performing the explained code translation and using the bounded model checker CBMC to establish preconditions for code minimisation [22]. The thesis author's contributions are detailed in Sec. 1.5.1.

Chapter 5 introduces and describes our program synthesis algorithm and its applications. We illustrate in Sec. 5.4 how a variety of verification and synthesis problems can be mapped to our engine, such as safety checking, bug finding and controller synthesis. We demonstrate experimentally that our algorithm compares favourably against both general purpose program synthesisers as well as specialised verification tools [28, 26, 2, 3]. The thesis author's relevant contributions in this area are listed in Sec. 1.5.2, Sec. 1.5.4, and Sec. 1.5.5.

In Chapter 6, we present our fully automated refactoring decision procedure, which employs the previously introduced program synthesis engine to construct provably correct, *semantics*-driven refactorings. We further present our Java Stream Theory JST, which reasons over the Java 8 Stream semantics on Java heap data structures with sufficient precision to establish refactoring preconditions. Based on both contributions

we present our tool *Kayak*, which implements the "Replace Loop by Java 8 Stream query" refactoring introduced in Sec. 1.3. The author's contributions to the work presented in this chapter are listed in Sec. 1.5.3.

1.5 Contributions

The contributions of the author of this thesis, referred to as "the author" in the following, are listed in the following with references to the respective research publications.

1.5.1 Remove Unnecessary Reflection

The engine presented in Chapter 4 is based on the CBMC bounded model checker [22]. The Java implementations of ANSI-C standard libraries, the translators from C to Java expressions and the constraint generation passing the equivalence problems to CBMC are solely the work of the author.

1.5.2 Danger Invariants

In [28], a program synthesis engine based on the Counterexample-guided inductive synthesis paradigm [88, 13] is presented. The purpose of this engine is the synthesis of danger invariants, which prove the presence of deep bugs. A program attempting to solve this program synthesis problem, called "Kalashnikov", was published in [29] without the author's contribution. Kalashnikov is implemented in the Python scripting language and implements the CEGIS paradigm. As a synthesis engine for new candidate programs both a genetic search strategy and CBMC's [22] symbolic execution engine are used. The verification engine is implemented using CBMC along with explicit and exhaustive counterexample search due to low performance of the CBMC back-end. The paper was rejected from relevant venues and was published on arXiv on 18th March 2015. The author of this thesis then joined the group with the goal to prove that the CEGIS paradigm was a viable strategy to synthesise danger invariant bug proofs. The author implemented the CBMC CEGIS module, which is an extension programmed into C++ and directly integrated in the CBMC code base. This new CBMC extension is solely the author's work and features the following improvements over the original Kalashnikov implementation:

Faster candidate solution verification: Direct integration in CBMC allows translation of candidate solutions to CBMC's GOTO instruction language before verifying. This reduced verification time to a point where this phase became a negligible portion of the synthesis time ($< 5\%$) and the explicit, exhaustive counterexample search was no longer necessary.

Biased GA crossover: The previous off-the-shelf genetic algorithm implementation was replaced by a custom implementation for the program synthesis domain. The new implementation features a biased selector function when selecting parents for a genetic crossover implementation. The function favours the combination of parents which solve disparate counterexamples reducing the risk of entering local minima, evidenced by the additional benchmarks solved between [29] and [28].

Information sharing for CEGIS engines: The symbolic execution and GA engine in Kalashnikov were completely independent and did not share information apart from found counterexamples. The author's new implementation transports counterexamples found by the symbolic execution engine into the genetic population of the competing GA back-end. Copies of such a candidate program replace a configurable portion of the GA's solution population. This allows the symbolic execution engine to guide the GA back-end out of local minima. The effect of this was significantly faster synthesis times as presented in [28] and compared to [29].

Additional benchmarks: The author repeated the experiments conducted with Kalashnikov and also extended the benchmark set with additional entries from the SV-COMP 2016 [93] software verification competition.

The improved danger invariants solver implemented using the new CBMC CEGIS module was accepted for presentation in [28].

1.5.3 Kayak: Safe Semantic Refactoring to Java Streams

The program synthesis engine presented in [28] is used in [27] to refactor redundant Java loops to equivalent Java 8 Stream constructs. The presented CBMC CEGIS extension, the front-end used to interpret refactoring problems, as well as the implemented abstraction and refactoring constraint generation are solely the work of the author. The author entirely programmed the presented refactoring tool "Kayak".

1.5.4 Program Synthesis for Program Analysis

The author's contributions listed in Sec. 1.5.2 and Sec. 1.5.3, combined with synthesis-related work by other authors [31, 32], were published as an extended journal version in [26].

1.5.5 Controller synthesis

Our work published in [2] and [3] enables the synthesis of digital controllers for continuous plants and state-space physical plants. The algorithm is based on the CEGIS engine presented in [28]. The author solely implemented the digital controller extension to the CBMC CEGIS module and significantly contributed to the digital controller synthesis constraints expressed in the C programming language in [2]. The author also significantly contributed to the completeness threshold controller synthesis constraint presented in [3].

The presented results are an alternative application of the synthesis engine published in [28] and demonstrate its wide range of applications.

1.5.6 Learning the Language of Error

The program visualisation engine published in [16] allows for the generation of an abstract graphical representation of programs as well as errors in programs or changes between program versions. This is particularly useful for visualising the impact of code refactorings. The author contributed significantly to the formal algorithms as well as the C++ implementation of the presented tool, of which the author remains the chief maintainer. The paper represents corollary work in the area of refactoring support and is not included in this thesis.

1.5.7 Assisted Coverage Closure

The coverage generation engine published in [77] explores the use of CBMC's symbolic execution engine to generate missing coverage in large industrial projects. It demonstrates that CBMC can be successfully employed to remove coverage gaps in a unit test suite. This is especially important before refactoring tasks to increase confidence in the correctness of the introduced changes.

This line of work is considered a side project to the author's automated refactoring efforts and is not included in this thesis. The author solely implemented the RapiCover extension presented, which automatically generates coverage queries for the FShell test vector generator.

Chapter 2

Preliminaries

2.1 CBMC and Symbolic Execution

Symbolic execution examines the semantics of a given program statically by computing a symbolic model of the program's possible states [64]. In this document we frequently refer to the bounded model checker CBMC [22], which employs symbolic execution for model checking. We give a brief overview of how CBMC analyses a program to provide the context for the model checking features we rely on in the following chapters. CBMC employs symbolic execution to map a program's semantics to a SAT formula which is satisfiable iff a certain property about the program semantics holds. This technique is used to find bugs in programs, such as an assertion violation or a null pointer dereference. Another common property to check using symbolic execution is the functional equivalence of two programs or functions under any possible input [41], which proves the soundness of refactorings and other code mutations. CBMC's symbolic execution by default limits the number of iterations for loops in the program, to produce a finite SAT formula in the context of possibly infinite loops. We summarise the specific steps by which CBMC converts an input program to an SAT instance in the following.

Unwind loop constructs: Loops implement the repeated execution of a list of instructions until a certain condition is met. CBMC handles loops by a process called "unwinding". It repeats the loop body a limited number of times and guards each body execution with the loop guard as illustrated in Fig. 2.1, with the original loop in Fig. 2.1a and the unwound statements in Fig. 2.1b. In the case of loops for which CBMC cannot determine an upper bound of necessary unwindings, users can specify a limit to instruct CBMC to disregard program paths containing more loop iterations in its analysis. CBMC optionally instruments an unwinding assertion at the end of its

loop unwindings, as seen in Fig. 2.1a, which CBMC also translates as a property to be verified. Unwinding assertion property violations warn the user that the chosen loop bound was not sufficient for all paths in the program.

```
int result = 1;
int i = 0;

while (i < 3) {
  result *= 2;
  i++;
}
```

(a) Original loop

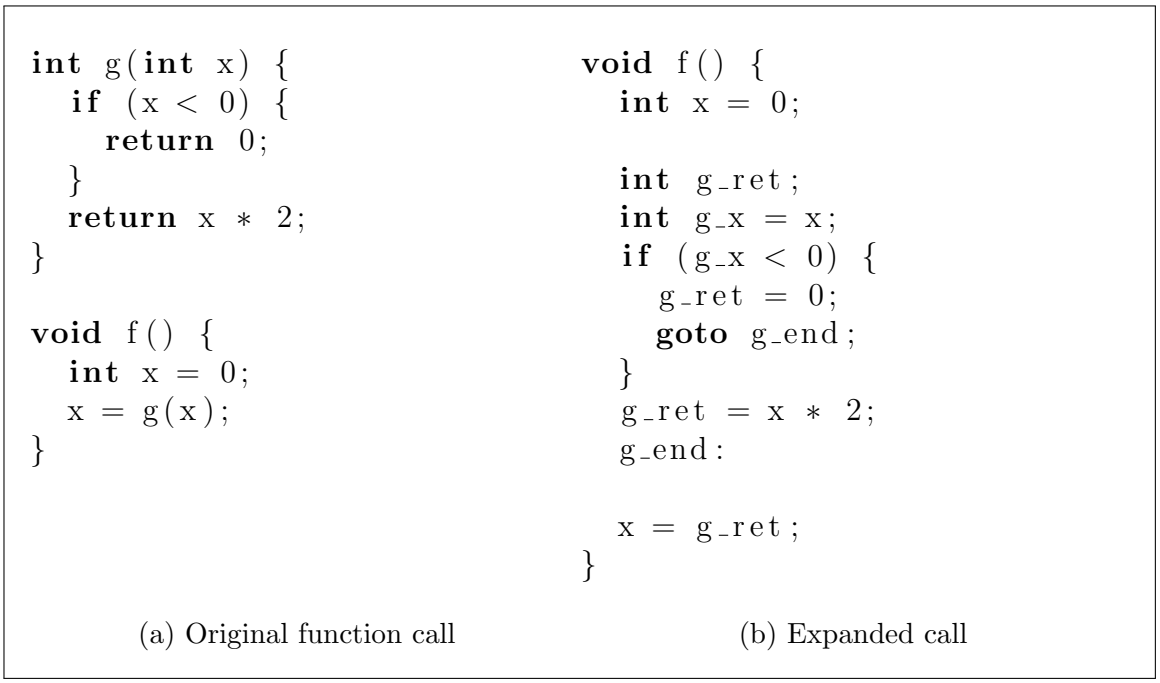
```
int result = 1;
int i = 0;

if (i < 3) {
  result *= 2;
  i++;
}

if (i < 3) {
  result *= 2;
  i++;
}

if (i < 3) {
  result *= 2;
  i++;
}

assert(i >= 3);
```

(b) Unwound statements

Figure 2.1: CBMC loop unwinding

An alternative to unwinding is computing corresponding safety invariants for the loops in the input program. We explain the concept of loop invariants briefly in Sec. 2.2. Using this approach, we can reason over properties of the input programs irrespective of the number of times its loop bodies are executed (cf. Chapter 6).

Expand function calls: CBMC inlines function calls in its expansion phase by replacing `return` statements by `goto` statements to the end of the function call, and potential return values by an assignment to a meta variable. This process is illustrated in Fig. 2.2, with the original function call in Fig. 2.2a and its replacement in Fig. 2.2b.



```
int g(int x) {
  if (x < 0) {
    return 0;
  }
  return x * 2;
}

void f() {
  int x = 0;
  x = g(x);
}
```

(a) Original function call

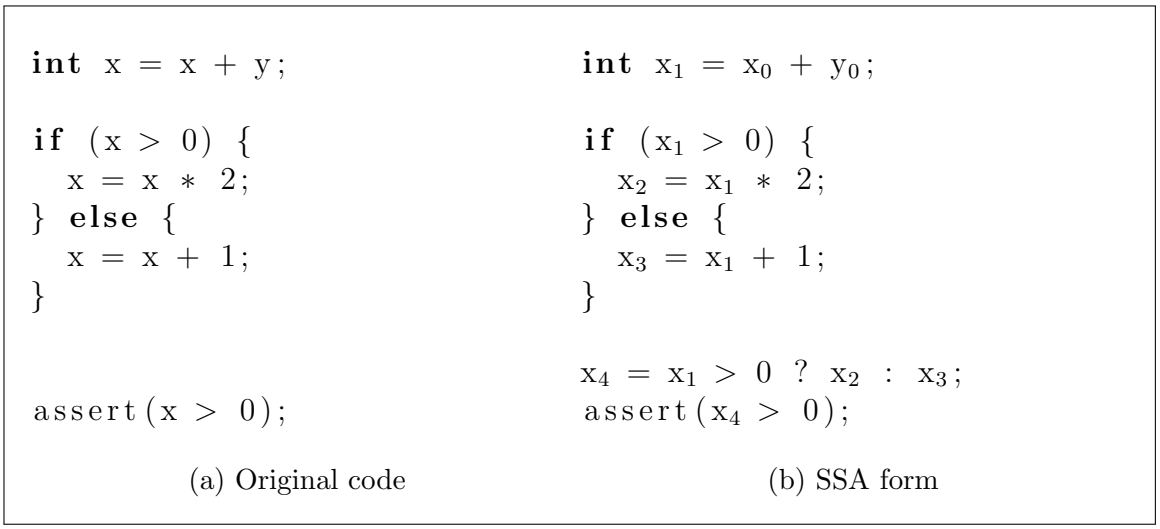
```
void f() {
  int x = 0;

  int g_ret;
  int g_x = x;
  if (g_x < 0) {
    g_ret = 0;
    goto g_end;
  }
  g_ret = x * 2;
  g_end:

  x = g_ret;
}
```

(b) Expanded call

Figure 2.2: Function call expansion

Recursive function calls are treated similar to loop constructs and are unwound up to a bound, asserting afterwards that the recursion does not continue deeper.

Convert to static single assignment (SSA): After the previous steps, a program can be transformed to static single assignment form (SSA), requiring that each variable in a program to be assigned exactly once. CBMC accomplishes this by introducing versions for each program variable depending on which assignment is applied to the variable. Fig. 2.3 illustrates this transformation with the original program in Fig. 2.3a and the SSA equivalent in Fig. 2.3b. SSA transformation is necessary to represent the model of sequential program statements in an unordered SAT instance. The variable versions allow the formula to express that e.g. the property in the assertion in Fig. 2.3a refers to the value of the variable x after it has been assigned a new value in the previous statements.



(a) Original code (b) SSA form

Figure 2.3: SSA conversion

Convert to bit-vector equations: The SSA program in Fig. 2.3b can be interpreted as two bit-vector equations:

$$\begin{aligned} P :=& x_1 = x_0 + y_0 \wedge \\ & x_2 = x_1 * 2 \wedge \\ & x_3 = x_1 + 1 \wedge \\ & x_1 > 0 \Rightarrow x_4 = x_2 \wedge \\ & x_1 <= 0 \Rightarrow x_4 = x_3 \\ A :=& x_4 > 0 \end{aligned}$$

In order to check that the property holds for the program, CBMC converts the formula $P \wedge \neg A$ to conjunctive normal form and passes it to a SAT solver. If the formula is unsatisfiable the program is safe. Otherwise CBMC uses the satisfying assignment provided by the solver to construct a counterexample trace.

CBMC exposes a rich API which allows the use of its bounded model checking functionality as a framework. Client applications can use this API to build decision procedures reasoning about the behaviour of input programs. Multiple contributions in this thesis rely on CBMC's framework, and we highlight at this point two key features which are frequently used when working with it: nondeterminism and assumptions.

Nondeterminism: A frequent use case in verification is testing a property of a program or function under every permitted input. For this purpose, CBMC allows

assigning a nondeterministic value to a variable. Doing so permits the underlying SAT solver to assign it any value within the variable's domain to expose a bug in the program. CBMC exposes this feature through many ways in its API, one of which is by assuming the return values of functions without a body are nondeterminstic. Fig. 2.4 illustrates this with a short code example, which verifies if any integer multiplied by the constant 2 is larger than the original number. CBMC will provide a counterexample for the illustrated property, since any negative integer number violates this constraint.

```
int nondet_int();

void f() {
  int x = nondet_int();
  int y = x * 2;
  assert(y >= x);
}
```

Figure 2.4: Nondeterminism example

Assumptions: CBMC considers every possible path through the input program with respect to the program's transition relation and the nondeterministic choices configured. In some verification scenarios, it is useful to restrict this selection, such as if certain traces are not interesting or outside the input specification of the program. For this purpose, CBMC provides the concept of assumptions, which instruct it to encode explicit constraints into the SAT formula, thus restricting its analysis to a subset of the paths in the program. As an example, in Fig. 2.4 it would be desirable to restrict the permissible inputs for the variable x to positive integer values to exclude obvious or uninteresting counterexamples. The CBMC API syntax achieving this is illustrated in Fig. 2.5. With this assumption in place, CBMC will provide a more interesting counterexample for x, which will violate the property by causing an integer overflow.

```
int nondet_int();

void f() {
  int x = nondet_int();
  __CPROVER_assume(x >= 0);
  int y = x * 2;
  assert(y >= x);
}
```

Figure 2.5: Assumption example

A detailed description of CBMC's remaining features and implementation is available in [22].

2.2 Loop Safety Invariants

We assume a generic loop with a pre- and postcondition, guard G, and transition relation T: $\{Precondition\}\ \mathtt{while}(G)\,T\ \{Postcondition\}$.

For such a loop, we can prove partial correctness, i.e. any terminating execution starting in a state satisfying *Precondition* reaches a state satisfying *Postcondition*, by finding a *safety invariant*, *Inv*, with the following properties:

$$\exists Inv.\forall x, x'.Precondition \rightarrow Inv(x) \wedge \tag{2.1}$$

$$Inv(x) \wedge G(x) \wedge T(x, x') \rightarrow Inv(x') \wedge \tag{2.2}$$

$$Inv(x) \wedge \neg G(x) \rightarrow Postcondition \tag{2.3}$$

In this formula, (2.1) ensures the safety invariant holds in the initial state, (2.2) checks that the invariant is inductive with respect to the transition relation, i.e., the transition relation maintains the invariant, and (2.3) ensures that the invariant establishes the postcondition on exit from the loop. This can be generalised to multiple, potentially nested, loops.

2.3 Java Native Interface

Compiled bytecode for the Java programming language runs within a virtual machine, independent of the underlying platform. JNI is a programming framework that

provides Java programs access to native machine code libraries. This feature is used to improve performance and enable the reuse of existing libraries.

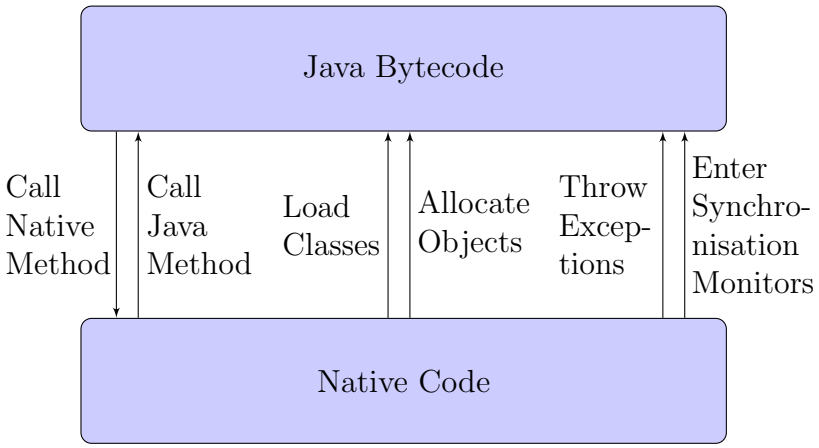


Figure 2.6: JNI features

The source language for these native libraries varies, and C or C++ is used in most cases. For the rest of this document, we thus focus on native code written in C. We refer to the combination of a Java program with JNI-enabled C code as JNI/C.[1] Fig. 2.6 illustrates the features accessible by JNI-enabled code, and we enumerate and explain a subset of these features with examples in the following.

Call a native method: Fig. 2.7 illustrates the syntax necessary to call a native C function from a Java program. The function `square` in this example is part of a library "libtest" which must be explicitly loaded before calling the method. The JNI interface for C programs exposes equivalent types for all Java types, such as the type `jint` in the example, which provides the same domain guarantees as the Java `int` type.

[1] http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html

```
class JNITest {
 static {
  System.loadLibrary("libtest");
 }

 native int square(int x);

 public void f(int x) {
  int y = square(x);
  int z = x * x;
  assert y == z;
 }
}
```

(a) Java

```
JNIEXPORT jint JNICALL
 Java_JNITest_square(JNIEnv *env,
                     jobject object, jint x) {
 return x * x;
}
```

(b) Native JNI/C

Figure 2.7: Native method call

Calling a Java method from native code: Similar to the example in Fig. 2.7, native methods are equally able to access and call methods in the Java virtual machine. Since the Java programming language supports features without first class equivalent in C, the JNI programming interface exposes data structures and utility functions in order to manipulate Java objects. Fig. 2.8 illustrates the process of calling a Java method from a native function.

```
class JNITest {
 static {
  System.loadLibrary("libtest");
 }

 public native void test(String param);

 public static void main(String[] args){
  JNITest obj = new JNITest();
  obj.test();
 }

 public void sayHello(){
  System.out.println("Hello world!");
 }
}
```

(a) Java

```
JNIEXPORT void JNICALL
 Java_JNITest_test(JNIEnv *env, jobject obj) {
  jclass JNITest = (*env)->FindClass(env, "JNITest");

  jmethodID sayHello = (*env)->GetMethodID(env,
                        JNITest, "sayHello", "()V");

  (*env)->CallVoidMethod(env, obj, sayHello);
}
```

(b) Native JNI/C

Figure 2.8: Call Java method from native

2.4 CEGIS paradigm

The Counterexample Guided Inductive Synthesis (CEGIS) [88, 13] paradigm describes a category of algorithms frequently used in the context of program synthesis. Algorithms implemented in the CEGIS paradigm are comprised of an iterative refinement loop and consist of a Synth and a Verif phase, as illustrated in Fig. 2.9. CEGIS algorithms traverse solution sets of exponential size, where a correct solution needs

to satisfy a constraint over an exponential domain. The algorithm performs well in scenarios where a limited subset of counterexamples from the domain is sufficient to constrain a solution that will pass for the entire domain.

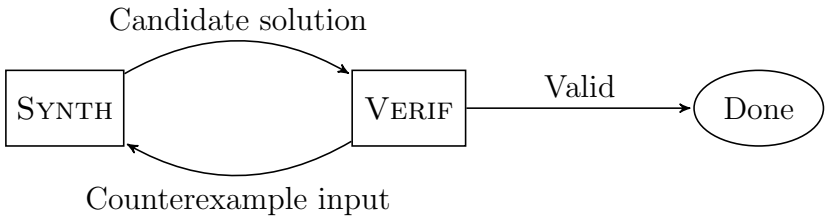


Figure 2.9: Abstract synthesis refinement loop

The algorithm starts with an empty or uninitialised candidate solution, The Verify phase checks a given candidate solution against the input constraint. If the candidate is valid, then the algorithm terminates. Otherwise a counterexample from the domain is provided for which the solution does not satisfy the constraint. These counterexamples are used in the Synth phase to refine the current candidate solution and guide the search through the solution set. In the case of program synthesis, the solution set is the set of all possible programs within a specified grammar, and the domain is the set of all allowed program inputs for which the synthesised program is expected to behave correctly.

The CEGIS paradigm only describes a family of counterexample guided refinement search algorithms. The actual algorithmic implementations are contained in the Synth and Verif phase algorithms and vary greatly depending on the solution set and domain over which the synthesis process is performed.

Chapter 3

Related Work

This chapter summarises relevant existing work for each chapter of this thesis, and along with Sec. 1.2 and Chapter 2 represents our assessment of the state of the art upon which this thesis seeks to progress.

3.1 Opaque library modelling

The work cited in this section is focused on the problem of modelling native or otherwise opaque program libraries. They represent competing modelling strategies to the one presented in Chapter 4.

There are alternative approaches for analysing native method calls in Java programs. In particular, any algorithm that abstracts library method calls is applicable in this scenario. Shafiei and van Breugel present a JNI extension to the Java model checker JPF in [84]. The plug-in passes any native method call to the underlying virtual machine, effectively executing it on the system. While this is a viable solution for many scenarios, the approach entails some significant drawbacks. Since the state of the JNI code is not modelled within JPF, symbolic values and other optimisations cannot be applied. The executed calls also have an impact on the system environment, which may not be readily reversible to explore a different path. Database applications and other persistence systems are particularly problematic. In this scenario, the verifier manipulates the database content during verification, which can yield unsound results.

Werner and Holger further present a summarisation approach for library functions [43] with an algorithm that automatically generates *transformation summaries* and *error triggers* from the library binary. The goal of the transformation summary is to specify how the program state is affected by the function call, which is expressed as a transfer relation. Error triggers serve as program state assertions and indicate a possibility of a program failure if satisfied at the call site. This information generates a

numeric program, which they verify using off-the-shelf numeric analysers. The system targets use cases where programs use well-established library functions. In particular, verification of the library function itself is not addressed.

Another translation approach to this problem is provided by Trudel et al. [95]. They provide full source-to-source translation from C to Eiffel by creating equivalent Eiffel models for all C language elements. Pre-compiled library functions are modelled manually.

Other systems abstract library calls by means of pre- and postconditions. Cousot et al. present such an algorithm with which they automatically infer necessary preconditions. They define *necessary* preconditions as conditions which, if violated, render the program always incorrect. This is a weaker condition than *sufficient* preconditions and leads to a sound, but incomplete abstraction. Overall, their approach outperformed comparable approaches by 9–21% precision [25].

Mariani and Pezzè further illustrate a run-time behaviour capturing approach to solve this problem with a system that monitors component libraries and collects run-time data during program executions. This data is then used to approximate the behaviour of the library and determine its properties [74]. Mariani, Pezzè, et al. also present a static approach to model extraction called "Static Extraction of Interaction Models (SEIM)" [73] that targets interactions between web services and clients. It produces a finite-state automaton that models the protocol requests sent by the client application. Interpreting native method calls in Java as a client-server protocol would, in a similar fashion, enable the extraction of models for Java verification.

Tan explores formal operational models for foreign function interfaces (FFIs) in [94]. This approach is hampered by the lack of formal specifications for systems falling into this category. Tan presents JNI Light (JNIL), which models a subset of JNI and provides explicit abstractions for interactions between high-level, garbage-collected languages and lower-level native languages. Tan proposes abstractions for handling a shared heap, cross-language method calls, exception handling, and garbage collection. Using JNIL with existing Java model checkers is subject to explicit integration and may provide weaker guarantees due to its abstractions than what the target model checker would be able to establish on regular Java code.

Tan and Li also present a static analysis algorithm in [70] to detect misuses of exceptions in the context of JNI. Their approach is targeted at specific misuse patterns of exceptions in native code and is implemented as an Eclipse plug-in. Other semantics, bugs or abstractions outside the misuse of exceptions are not covered.

Siefers, Morrisett and Tan also present a sandboxing approach to verify native code in their work on Robusta [87]. Instead of verifying arbitrary native code, they present a framework that entirely prohibits unsafe system modifications, confidentiality violations, and dynamic linking/loading. These restrictions allow for establishing security-related properties at the cost of a runtime overhead due to the sandbox. This work is extended by Sun and Tan in [92] where they present the tool Arabica. While Robusta is closely linked to an OpenJDK implementation, Arabica's two-layered implementation is portable between different VMs. Both Arabica and Robusta rely on restricting the functionality of the native code, and are not usable for arbitrary native programs.

3.2 Bug finding

Static bug finders that use techniques such as Bounded Model Checking (BMC) search for proofs that safety can be violated. They also have the attractive property that once an assertion fails, a counterexample trace is returned, to be inspected by the user [20], which represents proof that an assertion violation occurs. To construct such a danger proof, bounded model checkers compute underapproximations of the reachable program states by progressively unwinding the transition relation. The downside of this approach is that static bug finders fail to scale when analysing programs with bugs that require many iterations of a loop. The computational effort required to discover an assertion violation typically grows exponentially with the depth of the bug.

Notably, the scalability problem is not limited to procedures that implement BMC. Approaches based on a combination of over- and underapproximations such as predicate abstraction [21] and lazy abstraction with interpolants [75] are alo not optimised for finding deep bugs. This is because they can only detect counterexamples with deep loops after the repeated refutation of increasingly longer spurious counterexamples. The analyser first considers a potential error trace with one loop iteration, only to discover that this trace is infeasible. Consequently, the analyser increases the search depth, usually by considering one further loop iteration. This repeated unwinding suffers from the same exponential blow-up as BMC.

Danger invariants enable proving the existence of a bug without explicitly showing an error trace. This allows for more compact and intuitive proofs, which allow for more scalable analyses that do not suffer from false alarms.

Concerning the verification of temporal properties, a danger invariant for a loop with an assertion A essentially proves the CTL property $\models EF\neg A$ over the loop. While there exist CTL verifiers based on a reduction to exist-forall quantified Horn clauses [9, 8], we specialise the concept for finding deep bugs and describe a modular constraint generation technique over arbitrary programs instead of transition systems.

Another successful technique for finding deep bugs without false alarms is loop acceleration [67, 68]. This approach works by taking a single path at a time through a loop to compute a symbolic representation of the exact transitive closure of the path (an accelerator) and add it back into the program before using an off-the-shelf bug finder, such as a bounded model checker. Loop acceleration requires that each accelerated path be represented in closed-form by a polynomial over the program variables, which is not always possible. In contrast, danger invariants are complete such that a program has a corresponding danger invariant iff it has a bug. Loop acceleration could be used in concert with danger invariants, since if an accelerator can be found, then it is the strongest inductive fact about a loop making it a good candidate danger invariant.

3.3 Inductive program synthesis

Program synthesis is the mechanised construction of software that provably satisfies a given specification. Synthesis tools promise to relieve the programmer from thinking about *how* the problem is to be solved. Instead, the programmer only provides a compact description of *what* is to be achieved. Foundational research in this area has been exceptionally fruitful, beginning with Alonzo Church's work on the *Circuit Synthesis Problem* in the sixties [19]. Algorithmic approaches to the problem have frequently been connected to automated theorem proving [72, 66]. Recent developments include an application of Craig interpolation to synthesis by [51].

In the seminal paper, [46] describe Brahma as a program synthesiser for loop-free programs over bit-vectors. A key difference between our work and Brahma is that Brahma is designed to be used by a human operator guides the synthesis process, while our synthesiser is fully automatic. While Brahma uses a fixed set of components and encodes a program by finding appropriate "wiring" between the components, our tool finds SSA programs of arbitrary length. An important advantage of this encoding is that it does not require the user of the synthesiser to include all specification details, such as how many addition operations may appear in the program, which is key in enabling us to use the synthesiser as a black-box back-end for a plethora of use cases.

A recent successful approach to program synthesis is Syntax Guided Synthesis (SyGuS) [4], which supplements the logical specification with a syntactic template that constrains the space of allowed implementations. Thus, each semantic specification is accompanied by a syntactic specification in the form of a grammar. In contrast to SyGuS, our program synthesiser is optimised for program analysis according to the three aforementioned key dimensions.

Other second-order solvers are introduced by [44] and [9]. As opposed to ours, these are specialised for Horn clauses and the logic used is undecidable. [100] present a decision procedure for a logic related to the synthesis fragment, the Quantified bit-vector logic, which is a many sorted first-order logic formula where the sort of every variable is a bit-vector sort. It is possible to reduce formulae in the synthesis fragment over finite domains to Effectively Propositional Logic [76], but the reduction requires additional axiomatisation increasing the search space and defeating the efficiency we aim to achieve.

3.4 Refactoring

Cheung et al. describe a system that automatically transforms fragments of application logic into SQL queries [17]. Moreover, similar to our approach, the authors rely on synthesis technology to generate invariants and postconditions that validate their transformations (a similar approach is presented in [56]). The main difference with our work, other than the research goal, is that the lists they operate on are immutable and do not support operations such as remove. Capturing the potential side effects caused by these types of operations is one of our work's core challenges.

In syntax-driven refactoring engines, program transformation decisions are based on observations on the program's syntax tree. Visser presents a purely syntax-driven framework [96] intended to be configurable for specific refactoring tasks, but cannot provide guarantees about semantics preservation. The same holds for [24] by Cordy et al., [62] Sawin et al., [57] Bae et al., and [18] Christopoulou et al. In contrast to these approaches, our procedure constructs an equivalence proof before transforming the program.

Steimann et al. present Constraint-Based Refactoring in [89], [91], and [90] with an approach that generates explicit constraints over the program's abstract syntax tree to prevent compilation errors or behaviour changes by automated refactorings. This gives rise to a flexible framework of customisable refactorings, implementable through a refactoring constraint specification language (cf. [90]). The approach is limited by

the information a program's AST provides and favours conservative implementations of syntax-focused refactorings, such as *Pull Up Field*.

Fuhrer et al. implement a type constraint system to introduce missing type parameters in the use of generic classes (cf. [38]) and to introduce generic type parameters into classes which do not provide a generic interface despite being used in multiple type contexts (cf. [63]).

Raychev et al. present a semi-automatic approach where users perform incomplete refactorings manually and then employ a constraint solver to find a sequence of default refactorings, such as move or rename, which include the users' changes. The engine is limited to syntactic matching with the users' partial changes and does not consider program semantics [82].

Weissgerber and Diehl rely on meta information to classify changes between software versions as refactorings [99]. The technique aims to identify past refactorings performed by programmers but is not a decision procedure for automated refactorings.

O'Keffe and Cinnéide present search-based refactoring [79, 80], which is similar to syntax-driven refactoring. They rephrase refactoring as an optimisation problem by using code metrics as a fitness measure. The method optimises syntactical constraints and does not take program semantics into account.

Bavota et al. implement refactoring decisions in [7] using semantic information limited to identifiers and comments, which may differ from the actual semantics (e.g., due to bugs). Kataoka et al. also interpret program semantics to apply refactorings [58], but use dynamic test execution rather than formal verification, hence their transformation lacks soundness guarantees.

Franklin et al. implement a pattern-based refactoring approach transforming statements to stream queries [49] with a tool called LambdaFicator [36], which is available as a NetBeans branch. We compare Kayak against it in our experimental evaluation in Sec. 6.5.

Chapter 4

Exemplar: Remove unnecessary reflection

The Java Native Interface (JNI) offers an interface between native machine code with Java programs. Java programs that use the JNI are a challenge for static analysers. Reliable figures about the popularity of JNI are not readily available. As an approximate data point, there are around half a million C/C++ files referring to the JNI header files on GitHub[1]. This number suggests that JNI is used in a significant amount of open source software, so static analysis of such code is desirable.

We present a novel approach for automatically translating JNI code to Java models and test the hypothesis that these models enable the application of static analysers, such as model checkers and test case generators. Naive translation will yield undesirable properties in the translated code, such as an overuse of the Java Reflection API, so we apply an explicit automated refactoring to improve the code and remove unnecessary use of this API. We implement this approach in our tool called Jitsune[2]. We include a brief account of existing solutions to model native code in Java in Sec. 3.1, and we argue that this support can be measurably improved through the extraction of automatically generated Java models from the native source code. We substantiate this claim using experiments on a large set of open source Java benchmarks that access C library features using the JNI API. We use JPF [97] as a representative Java Model Checker and Evosuite [37] as a test case generator. Our contributions in this section and in the presented Jitsune tool are the following:

1. Translation of C to Java.

2. Java model of the JNI API system functions.

[1]Test search queries on http://www.github.com executed on 12/01/2016.
[2]A play on the Japanese word Kitsune, the shape-shifting fox.

3. Java model of the ANSI C standard library functions.

4. Post-processing and automated refactoring using the bounded model checker CBMC to remove unnecessary use of reflection.

5. Experimental analysis of Jitsune with Evosuite against a benchmark set of over 290,000 LOC.

4.1 Background

Motivating example: Java Pathfinder (JPF) is a Java model checker originally developed at the NASA Ames Research Center [97] containing a Java Virtual Machine implementation written in Java, and directly accepts Java bytecode input. In our experiments, we use the JPF core program (*jpf-core*) as well as the modules *jpf-nhandler* and *jpf-symbc*. The former allows concrete execution of JNI operations to approximate a model, and the latter implements symbolic execution and testing on top of the existing JPF architecture.

We illustrate the benefit JNI models offer for Java verification and test generation tools, such as JPF and Evosuite, using the example in Fig. 4.1. Both JPF and Evosuite aim to analyse code that uses the JNI, by concretising the symbolic state to a single concrete state, and then executing the JNI binary code with specific input values. This approach is ineffective for the function *resetDevice* in this example as the input space for the native operation has size 2^{32}. Since no real hardware device handles are set up beforehand, the only handle value which will cover the failing *"Don't reset working device!"* branch is $NULL\_HANDLE$ or $23,999$. Without a model for the JNI/C portion of the code, both JPF and Evosuite have no option other than to try every possible handle value to cover the special case. With a timeout of 300 seconds, both tools failed to do so in our experiment. JPF reports the program as safe, and Evosuite only achieves 67% coverage with its generated test cases. However, if we provide both tools with an equivalent Java model for the JNI code, JPF immediately reports an exception violation and Evosuite generates a correct test suite with 100% coverage. This example represents a best-case scenario for precise JNI models.

Models for native code, as generated by our approach, remediate two important liabilities of concretisation. First, concretising a symbolic analysis down to single values implies a substantial increase in computational complexity. To achieve completeness, the native code needs to be explicitly executed with every possible input value. Second, if the JNI code contains an internal state, executing these methods may lead

to changes in this state, which can cause inconsistent feedback from the perspective of the verification tool as well as unsound analysis results. An experiment measuring the effect obtained on large-scale real-world projects is described in Sec. 4.3.

Use cases: There are multiple use cases for the extraction of reliable models for JNI/C code. Since the models we extract are given as pure Java code, any Java model checker can integrate the system to obtain JNI/C support. Programmers who rely heavily on JNI for performance or compatibility reasons gain the opportunity to fully analyse the system. Currently, they need to manually abstract JNI method calls, which is error-prone and laborious. Better static analysis is desirable for mobile Java code, since distributing updates to these platforms is challenging. Our algorithm also allows migration of JNI-dependent Java programs to platforms that do not support JNI. The respective native code can simply be converted to Java so that the resulting program does not rely on native code.

Furthermore, our approach simplifies the modelling of Java runtime library functions, which are implemented in the native language. These are invoked using explicit JNI or equivalent native call mechanisms. Our approach can automatically convert these native C implementations to a Java model, which allows Java verifiers to analyse these functions automatically without the need for a manually constructed model.

Finally, our JNI models enable automated test case generators to increase the coverage achieved by providing a model for otherwise non-transparent JNI operations. To illustrate this specific point, we conduct an extensive experiment on the effects of our JNI models on Java test case generators in Sec. 4.3.

4.2 Model Extraction for JNI/C

The model extraction in Jitsune is comprised of two major components as illustrated in Fig. 4.2. The first phase translates the JNI/C source code to equivalent Java code by mapping each C expression to an equivalent Java expression, removing `goto` statements, which are not supported by Java, and translating pointer arithmetic.

The translated Java model is usable by model checkers and test case generators, such as JPF and Evosuite. The post-processing mode automatically refactors and optimises the generated model to simplify the analysis task for these tools. JNI system calls, which use string variables to look up Java class and method names, are translated by default using reflection. Reflection, however, makes the Java code less accessible for model checkers and should be avoided. Our post-processing phase rewrites these

```c
#define MAX_HANDLES 24000
#define NULL_HANDLE MAX_HANDLES - 1
#define OK_STATUS 0
#define ERROR_STATUS 1

 // Always false during tests.
_Bool existsHandle(int handle);

JNIEXPORT jint JNICALL
 Java_Devices_checkStatus(/* ... */, jint handle) {
  if (NULL_HANDLE == handle)
    return OK_STATUS;
  if (!existsHandle(handle))
    return ERROR_STATUS;

  // Use real hardware, never the case in tests...
}
```

(a) Native JNI/C

```java
class Devices {
 private static final int OK = 0;

 static {
  System.loadLibrary("devices");
 }

 static void resetDevice(int handle) {
  if (OK == checkStatus(handle)) {
    // Don't reset working device!
    throw new RuntimeException();
  } else {
    // Reset device...
  }
 }

 static native int checkStatus(int handle);
}
```

(b) Java

Figure 4.1: Synthetic Oracle example.

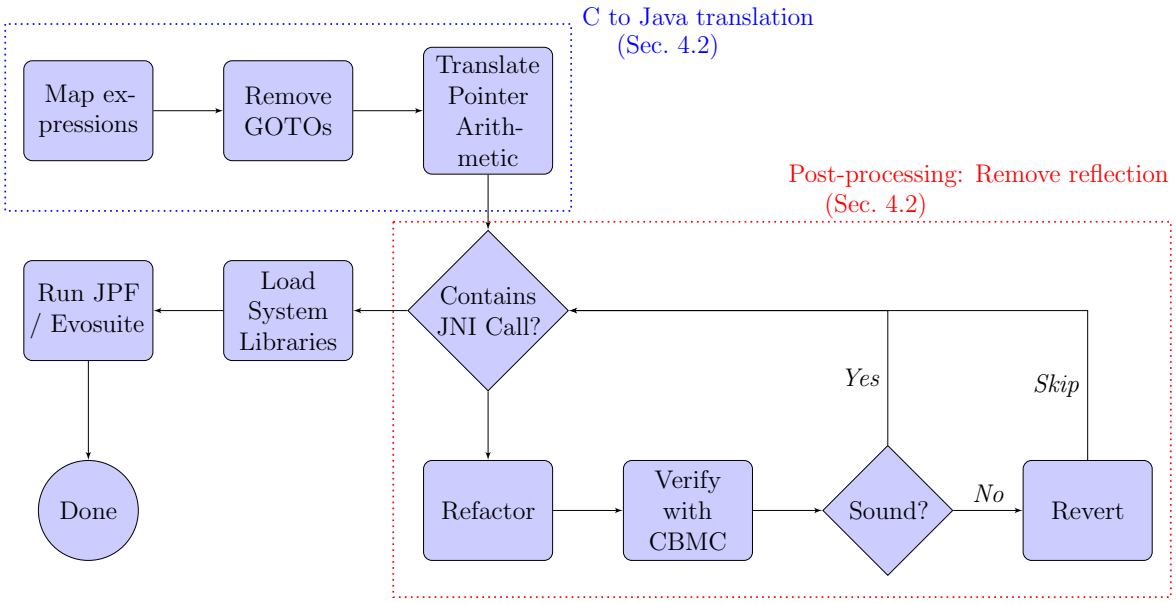


Figure 4.2: C to Java translation and JNI model creation

statements into equivalent code that does not use reflection and checks equivalence of the refactored model using CBMC. All processing steps in Jitsune are described in detail in the following.

Translation of C to Java: Fundamentally, creating our JNI Java models from C source code is a translation from one Turing-complete language to another. This translation poses many challenges resulting from the disjoint feature sets of the two languages. A large subset of the C language expression tree can be mapped to a direct equivalent in Java, such as function calls, operators, and declarations. Fig. 4.3 gives an example of such an expression-based mapping from C to Java.

We further illustrate translation from C features that are not directly supported in the Java language. Since both languages are equally expressive, every C feature can be modelled by a sufficiently complex Java model.

Pointer arithmetic: Most translated models do not explicitly maintain a C memory model. C types are translated to equivalent Java types and managed in the Java virtual machine's memory model. This process is sufficient to obtain equivalent behaviour, and provides optimal performance for Java analysers. We deviate from this approach only in the case of C pointer arithmetic. Pointer arithmetic allows read/write access to arbitrary locations within an object. Instead of analysing the precise semantics of such programs, which is undecidable in the general case, we create an explicit

```c
void main(int argc, char *argv[]) {
 char data[] = { 1, 2, 3 };
 int sum = 0;

 int i = 0;
 for(; i < sizeof(data) ; ++i) {
  sum += data[i];
 }

 puts("Done\n");
}
```

(a) Native JNI/C

```java
class main_container{

 static void main(String[] args){
  byte[] data = { 1, 2, 3 };
  int sum = 0;

  int i = 0;
  for(; i < data.length; ++i) {
   sum += data[i];
  }

  System.out.print("Done\n");
 }
}
```

(b) Java

Figure 4.3: Mapping simple C expressions to Java.

representation of the C memory model and move all variables that could be accessed using the pointer into this model. The C memory model knows both heap and stack memory categories [1]. We create a Java `byte` array for each with pre-configured maximum stack and heap sizes.

Theorem 4.2.1 (C and Jitsune *memory* equivalence)**.** *The original C program's heap contains a memory block m iff the translated Jitsune program's heap model contains an equivalent memory block m'.*

Proof. In the C language, every block of memory on the heap must be allocated using the `malloc` family functions [1]. Our algorithm replaces each `malloc` family function invocation by a call to a matching model function `Stdlib.malloc`. We assume that every address returned by `malloc` is a positive integer. For the case where memory allocation fails, both `malloc` and `Stdlib.malloc` return a canonical representation of `nullptr` and are trivially equivalent.

Allocated memory blocks in Jitsune are stored in a single `byte` array. We assume this array is larger than the maximum byte address on the original C heap. For every pointer i referring to a byte on the heap, there exists an element $heap[i]$ on the Jitsune heap. Jitsune maintains a tuple for each memory allocation, storing its index and size. In this system, for every allocation (p, s) by `malloc`, where p is the memory pointer and s is the size of the allocated block, there exists an equivalent allocation in Jitsune (i_p, s) with array index i_p and size s in the heap `byte` array. Equally, for every allocation by `Stdlib.malloc` (i, s), there also exists an allocation of (p_i, s) in the original C program.

Thus, both the original code and its translation will always maintain equivalent heaps and provide pointer representations to equivalent memory blocks. □

Fig. 4.4 demonstrates a translation of C pointer arithmetic to Java. We avoid wrapping other variables since this represents an unnecessary performance impairment.

Goto statements: `goto` statements are part of the Java bytecode language but are not exposed in the Java source code language. Our goal is to generate Java source code, which requires modelling this feature using source-level Java. To emulate `goto` semantics, we combine the backwards edge of a `while` loop with the Java `switch` statement's forward edge to arbitrary positions marked with `case` labels. The entire body of a translated function is wrapped inside such a nested `switch` statement, as illustrated in Fig. 4.5. When translating from C to Java, our algorithm transforms the program's control flow graph as illustrated in Fig. 4.6. It replaces `goto` statements

```
void run() {
 int *v = malloc(2u);

 int *p = (int *)v;

 *p = 0;

 ++p;

 *p = 1;
}
```

(a) Native JNI/C

```
public static void run(){
 Pointer<Void> v =
   Pointers.malloc(2u);

 Pointer<Integer> p = Casts.cast(
   Types.getInteger(), v);

 Operators.equals(
   Opertors.dereference(p), 0);

 Operators.preIncrement(p);

 Operators.equals(
   Opertors.dereference(p), 1);
}
```

(b) Java

Figure 4.4: Pointer arithmetic in the JNI model.

by an assignment to *label* and a subsequent `continue` statement. We prove the equivalence between these constructs in theorem 4.2.2.

Theorem 4.2.2 (C and Jitsune *goto* equivalence)**.** *The Jitsune `goto` replacement has the same successor nodes in the control flow graph as the original C `goto` under any possible execution.*

Proof. For this proof, we demonstrate that for every execution of a `goto` statement in the original C code there exists an execution of the Jitsune `goto` replacement, which, for the same original state, will lead to the equivalently labelled statement in the subsequent program state. We prove this by showing that both code constructs only have one possible execution trace which leads to a statement with the `goto`'s associated label. We assume that any loop in the C function body has been replaced by conditional backward `goto` statements, which is a transformation pre-applied by CBMC [22].

The first conjunct of this proof is trivially true since the C `goto` statement is defined as an unconditional jump to the statement with the associated label, as illustrated in Fig. 4.6.

```
static void f(){
 int label = 0;
 while(label >= 0) {
  switch(label) {
   case 0:
    // ...
   case 1:
    // ...
   default:
   label = -1;
  }
 }
}
```

Figure 4.5: Wrapped function body.

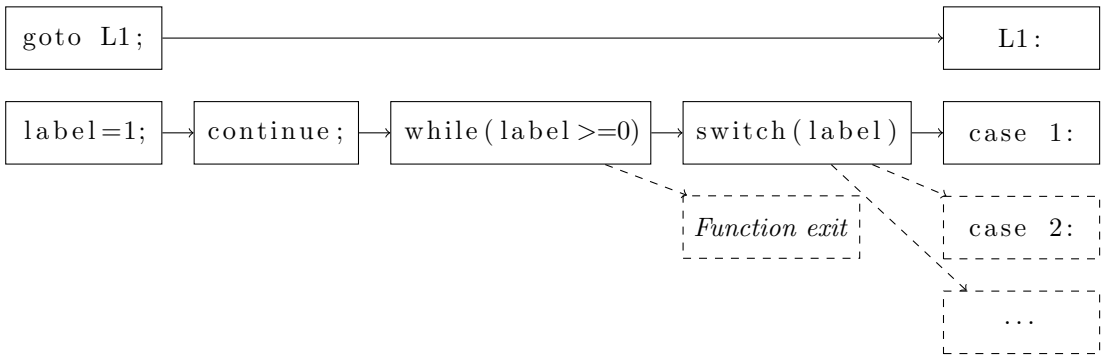


Figure 4.6: C to Java `goto` control flow graph node transformation.

For the second conjunct we assign every label in the C function to translate an integer $x > 0$. In the translation we wrap the C function's original body in a `while-switch` wrapper illustrated in Fig. 4.5. It is syntactically legal to replace every labelled statement of the C function with a `case x:` label of its associated x. The C `goto` statement itself is replaced by the statements `label=x; continue;`. Since loops are replaced from the user code by CBMC, the successor node in the control flow graph of the `continue` statement is `while(label >= 0)`. Since $x > 0$, the loop guard holds and is succeeded by `switch(label)`. The Java `switch` semantics guarantee that its successor will be the statement labelled with `case x:`. Since the variable `label` is assigned to x and never changed before reaching label x, this is the only possible execution. This sequence is illustrated in Fig. 4.6, and we present extended examples of this transformation in Appendix B. □

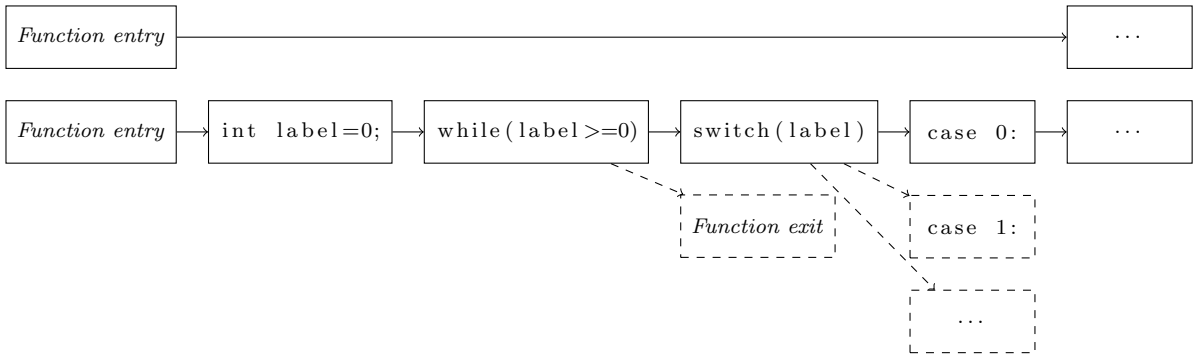


Figure 4.7: C to Java function entry transformation.

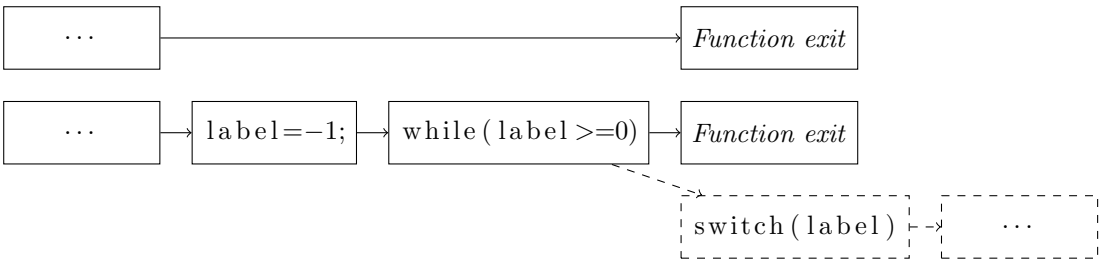


Figure 4.8: C to Java function exit transformation.

Initialisation rules: Unless explicitly initialised, local variables in C programs have a nondeterminate initial value. This is not the case in Java where all variables are initialised using appropriate default values. To match the C behaviour, we explicitly assign nondeterministic values to uninitialised variables in the translated models. Nondeterministic values are obtained using the API of the model checker, as in the case of JPF. If no such API is available, the default fallback is the *java.util.Random* class.

Finalisation: Variables allocated in stack memory are subject to deterministic finalisation in C. Accessing a variable outside of its lifetime scope results in undefined behaviour. Whenever Jitsune leverages Java heap objects to simulate stack variables, it uses a *try-catch* statement to make sure the respective memory is freed as soon as the variable is out of scope. This allows Jitsune to throw an *UndefinedBehaviourException* whenever memory is accessed out of scope. Thus, model checkers, such as JPF can detect and report undefined behaviour caused by illegal memory accesses. Fig. 4.9 provides an example of this feature.

ANSI C standard library: The ANSI C standard library is a vast set of utility and system functions. We explicitly model all standard library functions used in our

```
void run() {
 int *p = 0;

 {
  int i = 0;
  p = &i;
 }

 // Undefined behaviour
 int j = *p;
}
```

(a) Native JNI/C

```
public static void run(){
 Pointer<Integer> p = make_null();

 try { Memory.startBlock();
  Pointee<Integer> i = wrap(0);
  Operators.equals(p, ref(i));
 } finally { Memory.endBlock(); }

 // UndefinedBehaviourException
 int j=unwrap(deref(p));
}
```

(b) Java

Figure 4.9: Finalisation and undefined behaviour.

benchmarks. The Java runtime library provides the facilities to implement all features accordingly, but mapping specific ANSI C function signatures requires a non-trivial model. Such a model is integrated into our implementation of Jitsune.

Model post-processing: JNI offers an API for accessing Java program elements, such as classes or methods, for invocation from the JNI/C program. Classes and methods need to be looked up by name and are retrieved as meta-objects, which can be invoked later. The precise behaviour of the string-based class and method lookups can be reproduced in Java using Java's reflection API. Fig. 4.10 illustrates a reflection-based translation of a JNI system function call.

However, reflection is a non-trivial feature and remains challenging for Java analysers to implement, so our model creation algorithm attempts to avoid reflection. We observe that the class and method names in the example of Fig. 4.10a are constants. They will always result in the same method invocation under any execution. We, therefore, replace the reflection-based code by an explicit method call, as illustrated in Fig. 4.11. To apply this optimisation, we need to show that the string values for the class and method names will remain the same for any run of the program.

We obtain and verify this invariant using CBMC, which is a bounded model checker for C and C++ [22]. Among other features, it performs verification of program properties specified in the form of assertions up to a user-specified bound. If an assertion can be violated by any program input, CBMC will detect this and provide

```
jclass clazz = (*env)->FindClass(env, "JNITest");
jmethodID sayHello = (*env)->GetMethodID(
                                env, clazz, "sayHello", "()V");
(*env)->CallVoidMethod(env, obj, sayHello);
```

(a) Java

```
Class<?> clazz = Class.forName("JNITest");
Method sayHello = clazz.getDeclaredMethod("sayHello");
sayHello.invoke(obj);
```

(b) Native JNI/C

Figure 4.10: JNI system call translation using reflection.

```
((JNITest)obj).sayHello();
```

Figure 4.11: JNI system call translation using an explicit method call.

a trace leading to the violation. We first use CBMC to generate an execution trace, from which we extract a candidate class and method name. We use this candidate to refactor the JNI system call into an explicit method call. CBMC then verifies the two models always call the same operation under any input. This process is illustrated in Fig. 4.2. It is important to note that the equivalence guarantees this refactoring provides are limited by CBMC's verification guarantees. CBMC verifies equivalence only up to a user-specified maximum loop unwinding. Future work could expand on this limitation by relying on different model checking software to establish the equivalence property.

Explicit JNI model: Another component required for our approach is an explicit model for the functions provided by the JNI API in Java. The framework provides access to over 200 individual operations ranging from buffer allocation to throwing Java exceptions[3]. Our current implementation is incomplete but spans all operations necessary to model our benchmarks given in Table 4.1.

Limitations: Our current approach has two significant limitations, which reduce precision. First, during the automatic optimisation described in Sec. 4.2, we only analyse the JNI/C portion of the program to establish that the called operations are invariant. Although rare, it may be the case that the value of the class and function name strings depend on the associated Java model. One example of such a situation is given in Fig. 4.12. The method to be called on line 23 is determined by the return value of the Java method call on line 10. Since we are using CBMC as a pure C model checker for our invariant check, we have no model available for the result of the *getMethodCall* operation and must assume its return value is not constant. This issue could be addressed by passing the Java code to the model checker to extend the invariant check beyond the boundaries of the JNI framework.

A second limitation is rooted in the fact that our algorithm is specifically designed to optimise the direct usage of the Java Native Interface. Other interfaces or custom wrappers are not considered. Fig. 4.13 gives a trivial wrapper operation for a Java method call from JNI/C that will not be optimised by our algorithm. Addressing such constructs is outside of the scope of this contribution.

[3] `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html`

```
jclass clazz = (*env)->GetObjectClass(env, obj);

jmethodID getMethodName =
 (*env)->GetMethodID(
  env, clazz, "getMethodName", "()Ljava/lang/String;");

jstring methodName = (jstring)
 (*env)->CallObjectMethod(env, obj, getMethodName);

const char *strzMethodName =
 (*env)->GetStringUTFChars(env, methodName, 0);

jmethodID operationToBeCalled =
 (*env)->GetMethodID(
  env, JNIExample, strzMethodName, "()V");

jstring finalResult = (jstring)
 (*env)->CallObjectMethod(env, obj, operationToBeCalled);
```

Figure 4.12: Method and class name depend on the Java method result.

```
int JNI_CALL_COUNT = 0;

void my_jni_call(JNIEnv *env,
 jobject obj, jmethodID methodId) {
  (*env)->CallVoidMethod(env, obj, methodId);
  ++JNI_CALL_COUNT;
}
```

Figure 4.13: Wrapped Java Native Interface.

4.3 Experimental Evaluation

We experimentally test our claim that the JNI models generated by Jitsune improve the static analysis of Java programs using JNI/C. Specifically, our experiments show that exchanging Evosuite's internal model by JNI models generated by Jitsune improve class statement coverage by up to 80%.

4.3.1 Selection of Benchmarks

We use a set of 14 real-world Java software packages with JNI dependencies on four different native libraries as a benchmark suite. The four native libraries are SQLite, Heartbeats, NanoVG, and the JNI μ-Benchmarks set. The programs were selected randomly from GitHub using its search API (Table 4.1), and the total size of all benchmarks is over 290,000 LOC (before translation).

We provide both Jitsune as well as our benchmark set for download at `http://www.cprover.org/refactoring/jitsune.tar.gz`.

4.3.2 Experimental Setup

Evosuite is an automatic test case generator for Java projects. It employs a dual strategy consisting of both a search-based component and a dynamic symbolic execution engine (DSE). This combined approach makes it an interesting and challenging candidate for our experiments. We expect our JNI models to benefit purely symbolic tools most, since the flow-sensitive analysis cannot gather any information from JNI methods, requiring their over-approximation. Evosuite, on the other hand, does not rely alone on information retrieved from the Java source code, but can also employ concrete execution and evolutionary search combinations to improve its test suites [39]. Evosuite has a built-in model for handling JNI, which concretises the symbolic execution to single traces and executes the native binary code with explicit values.

We compare our generated JNI models against its built-in approach. We run Evosuite for each class in the benchmarks with its built-in model and with a JNI model generated by Jitsune. We only load the Jitsune JNI models if Evosuite executes a class that either contains a native method or has a dependency on a class with native methods. We measure the number of classes in the dependency chain between a specific class and a native method as "Distance to JNI", illustrated in Fig. 4.14. For classes with large Distance to JNI, the probability that it uses the model decreases. We thus define a cut-off distance after which we do not load Jitsune's model to avoid

Name	Size (LOC)	Description
SQLite	116,300	Relational Database Management System contained in a single C library. SQLite stores data in file-based databases and is popular especially in embedded environments. The library includes an extensive test suite. [4] [5]
Wakandan FYP	24,277	Marketplace simulation application. Illustrates use and effectiveness of Trust Models against real-world Attack Models. [6]
JCompoundMapper	21,860	Library for fingerprinting (decomposition) of chemical compounds. Features exporting options for data mining toolkits. [7]
MZDB Access	21,295	Java library for reading and querying mzDB files. mzDB (Mass Spectrometry SQLite Database) is a file format developed by the Proteomics French Infrastructure (ProFI). [8]
Pacioli	16,859	An accounting and project management system. [9]
Paysup	14,566	Personal desktop application for the creation of payment files for electronic banking. [10]
jjb	14,498	A bot for the Jabber XMPP instant messaging service. [11]
PMViewer	12,735	Management application for archived myBB private messages on PC and Mac. [12]
CSC202	12,466	A credit card processing database developed for testing purposes. [13]
sfscraper	12,146	HTML scraping utility. [14]
WebBot	12,144	Web crawler capable of parsing and downloading various web-related data types. [15]
JNanoVG	7,044	Java wrapper for NanoVG, a small antialiased vector graphics rendering library for OpenGL using the GLESv2 rendering library. [16] [17]
Heartbeats Simple	2,429	Low-level performance monitoring and analysis framework. [18] [19]
JNI μ-Benchmarks	1,648	Academic benchmark set for JNI performance analysis. [20]

Table 4.1: List of benchmarks.

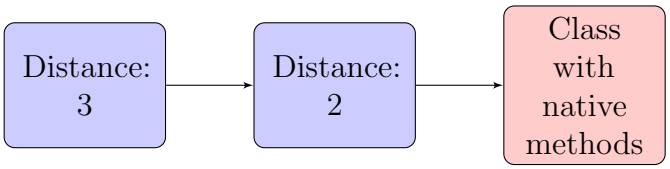


Figure 4.14: Distance to JNI.

Name	Increase
SQLite	60.17%
NanoVG	100.00%
Heartbeats	67.63%
JNI μ-Benchmarks	0.00%

Table 4.2: Results per native library. (cut-off=3)

unnecessary overhead. We present both coverage increases for classes at each Distance to JNI as well as the overall coverage improvement for all classes up to a maximum distance to JNI. We run Evosuite with a fixed seed value to improve reproducibility and set a time limit of 300 seconds per class.

4.3.3 Results

Table 4.2 groups the average coverage gains in the Java projects by their respective native library dependency. In the case of NanoVG, the coverage increase is 100%, since Evosuite is unable to generate *any* coverage without our models. This is because NanoVG relies on object handles passed to its native functions, and invalid handles crash the program. Evosuite's default JNI model, effectively guessing its input parameters, keeps crashing the VM and is unable to generate any coverage. Using the models generated by Jitsune, the Java program becomes trivially coverable, and we achieve full statement coverage. This represents a best-case scenario for Jitsune.

Due to our random selection strategy, the benchmarks also contain a worst-case scenario for Jitsune. The JNI μ-Benchmarks are a performance test environment for JNI and use deterministic inputs for its native method calls. Evosuite's explicit execution strategy represents an ideal approach for this scenario since all input parameters that will generate the necessary coverage are statically known and constant. Jitsune cannot generate any additional coverage in this scenario, which results in a 0% increase when using the generated model.

4.3.4 Discussion of Results

Our experimental results suggest that the Distance to JNI of a class significantly affects the benefit of the JNI model generated by Jitsune. For benchmarks with 100,000 lines of code and more, the generated JNI models are equally large and slow down Evosuite's dynamic symbolic execution process, thus generating less coverage in the same amount of time. As an example, the SQLite C source code is comprised of 116,300 lines of C code, which is translated by Jitsune to 299,612 lines of Java code. Since the Java projects using SQLite via JNI only include 10,000 lines of code on average, the generated model significantly increases the complexity of the full Java model. This performance penalty is offset by the fact that Evosuite can traverse the JNI functions more precisely using Jitsune's generated model. As illustrated in the motivating example in Sec. 4.1, without our JNI model Evosuite effectively needs to guess inputs that will result in return values necessary to cover remaining code sections. With the JNI model, Evosuite's dynamic symbolic execution can directly determine the necessary inputs to achieve the desired outcome.

The negative impact of the increased model size and the positive impact of increased precision for the JNI model balance each other and represent a trade-off for the coverage increase Jitsune provides. Classes that never call a JNI operation do not benefit from the more precise symbolic execution, and only suffer the performance impairment of a large JNI model loaded into Evosuite. Additionally, even if a class has an indirect dependency on a JNI method, our experiments show that the likelihood of coverability of a code section depending on a JNI output decreases as the Distance to JNI increases. We call this phenomenon the "Shielding" effect, since classes with high Distance to JNI

[4] http://www.sqlite.org
[5] https://bitbucket.org/almworks/sqlite4java
[6] https://github.com/wakandan/FYP
[7] https://github.com/fortiema/jCompoundMapper
[8] https://github.com/mzdb/mzdb-access
[9] https://github.com/nathanvander/pacioli
[10] https://github.com/davsva/Paysup
[11] https://github.com/stylesuxx/JJB
[12] https://github.com/Saadtronics/PMViewerNew
[13] https://github.com/rechner/CSC202CreditCardProcessor
[14] https://github.com/andreasaronsson/sfscraper
[15] https://github.com/SnakeDoc/WebBot
[16] https://github.com/memononen/nanovg
[17] https://github.com/chriscamacho/Jnanovg
[18] https://github.com/libheartbeats/heartbeats
[19] https://github.com/libheartbeats/heartbeats-simple
[20] https://github.com/qzan9/jni-ubmk

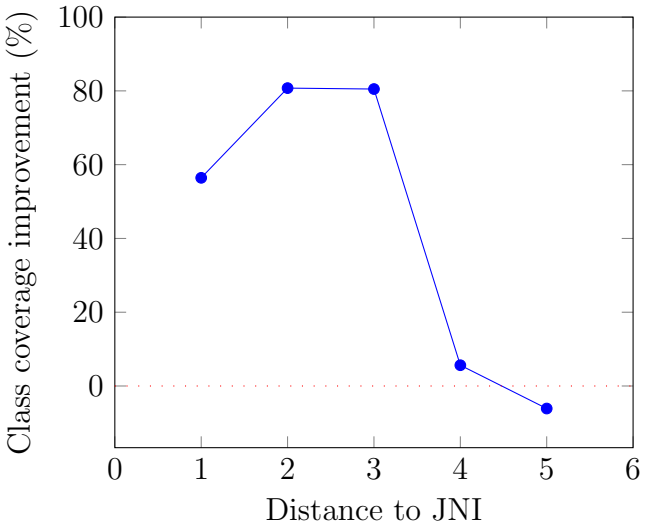


Figure 4.15: Average coverage improvement per Distance to JNI.

are shielded from the direct impact of JNI results by the abstraction the intermediate classes provide.

Fig. 4.15 quantifies the coverage gained for classes with a certain Distance to JNI. For our benchmark set, the average coverage benefit sharply declines after Distance 3. For classes with 4 or more intermediate classes to a JNI call, we obtain limited improvements from the precision of the JNI model compared to the performance impairment caused by the increased code size. Fig. 4.16 highlights the coverage benefit achieved using Jitsune when applied to classes with Distance to JNI up to a given threshold. The results over the 14 projects in our benchmark set suggest that Jitsune's JNI models should be applied for classes with a maximum Distance to JNI of two or three. At this limit, Jitsune achieves a coverage improvement of 80%. Classes beyond this cut-off do not benefit enough to justify the increased complexity caused by the size of the model.

4.4 Threats to validity

We identify four threats to the validity of our claims.

Translation Bugs: Jitsune is experimental research software, and bugs in Jitsune can lead to JNI models that are not equivalent to the original JNI/C source code. This might increase coverage artificially and is a threat to the claim that our approach improves static analysis of such programs. When used with Java model checkers, bugs in the translation result in unsound verification results. This threat is partially

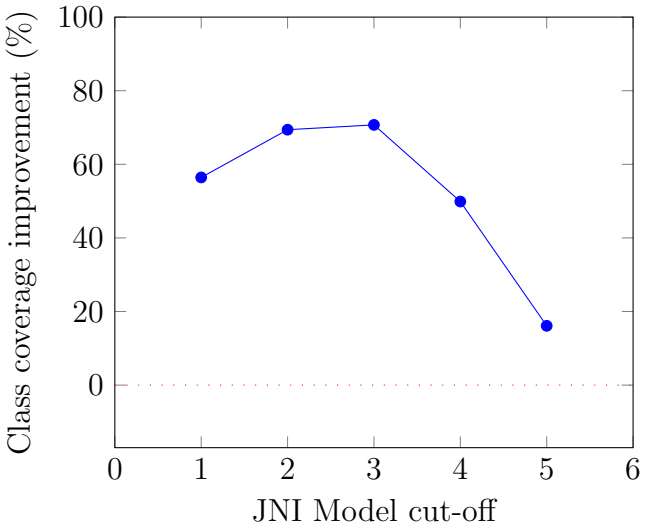


Figure 4.16: Average coverage improvement per distance cut-off.

mitigated by our evaluation using Evosuite, which verifies the generated test vectors against the original program and removes them if they do not cover the expected code section. Potential bugs in Jitsune could be observed as reduced test coverage.

We further mitigate this threat by running the unit tests of each benchmark with the generated JNI model. For projects such as SQLite, which comes with an extensive set of test suites that achieve 100% branch and MC/DC coverage, this provides reasonable assurance that the generated models are correct. Other projects among our benchmark set do not feature comparable test suites, however, and are susceptible to model equivalence bugs.

Choice of benchmarks: A second threat is that our benchmarks are not representative for JNI/C programs. This critique is addressed using our large-scale experimentation with a code base of over 290,000 LOC of open-source programs. However, it remains possible that closed-source projects, which are inaccessible to us, exhibit systematically different usage of JNI that would reduce the measured coverage benefit compared to what we observe on open-source projects.

Choice of analysers: A third threat is the limited set of analysers (JPF and Evosuite) used in our evaluation. Other Java analysers might obtain less benefit from our JNI/C models or might be unable to analyse the Java code we generate.

Choice of metrics: The primary metric for the claim that our approach improves the analysis of Java programs using JNI/C is test coverage. The validity of test

coverage as a metric for the utility of software testing is disputable [53], and the standard critique of such metrics applies in our case. Generated test coverage has the benefit of enabling a straightforward quantitative measure of the benefit provided by our models. Another way to draw such a quantitative improvement comparison would be to use benchmarks with a large set of known bugs and measure the number of additional bugs exposed with the model present.

Distance to JNI: The Distance to JNI is an abstract measure introduced in this chapter to estimate the benefits expected from applying our model. While this measure is congruent with our data and reliably predicts the impact of the model in our experiments, other measures may be better suited. Distance to JNI is a purely syntactic metric, and other metrics could instead measure how frequently a JNI method is invoked from a given class as an indicator of whether or not to apply our abstract model.

Chapter 5

Inductive Program Synthesis

In Chapter 4, we explored the use of bounded model checking for properties relevant to software engineering and refactoring decisions. In the provided examples it was sufficient to assert a given property over the input program and then perform a well-defined mutation of the program code. However, this is not sufficient if the resulting mutation is not predetermined, but needs to be selected from a large set of possible mutations. An example of this situation is the "Loop to Java 8 Stream" refactoring illustrated in Sec. 1.3. Algorithms applying this refactoring need to synthesise a query from the Java 8 Stream grammar which is equivalent to the original code. Proving that a given Stream query is equivalent to the original code can be mapped to a Propositional Satisfiability (SAT) instance using bounded model checking, as explained in Sec. 5.2 and 6. However, generating a query Q equivalent to the original code C introduces an additional quantifier alternation: $\exists_Q.\forall_x.Q(x) = C(x)$. Fundamentally, constructing query Q presents a program synthesis problem.

In this chapter, we solve this synthesis problem by proposing a general-purpose *program synthesis* framework. The chapter is based on our conference papers [28, 2, 3] and our journal publication [26]. The journal paper and this thesis use the synthesis problem definition and specification language introduced in "Using Program Synthesis for Program Analysis" [32]. The initial prototype of our program synthesis engine was called "Kalashnikov" and was implemented by Matt Lewis. Kalashnikov is considered obsolete and was replaced for our subsequent publications by the program synthesis framework implemented and presented in this thesis.

Our present framework allows implementing new program synthesis use cases by only providing a description of the corresponding program proofs. For a problem to be solved with our framework, it must be expressible in a fragment of second-order logic with restricted quantification, which we call the *synthesis fragment*. We show that the

synthesis fragment is general enough to capture many such problems by providing instantiations of our framework for the following diverse set of tasks:

- Safety – none of the assertions in the program can fail.
- Danger – at least one of the assertions can fail.
- Refactoring – making structured changes to existing code that improve its non-functional properties while leaving its externally observable behaviour unchanged (cf. Chapter 6).

To solve the problems expressed in the synthesis fragment, we built a novel program synthesis engine. We describe in the following how our engine differs from other general-purpose program synthesisers in three dimensions (identified as the three key dimensions in program synthesis by [45]).

Expression of user intent: Our specification language is *GOTO*, the intermediate representation of the bounded model checker CBMC [22], which results in *concise specifications* of program properties. *GOTO* contains only 18 instructions in CBMC's current version 5.7, chief among which is its namesake conditional `goto` expression to implement loops and branches. Despite a limited instruction set, *GOTO* instructions operate on a rich grammar of expressions similar to C. *GOTO* programs can thus represent C program semantics in a succinct and intuitive manner. Using our tool to implement a program synthesis use case only requires providing a generic specification of the problem to solve. Our experiments show that this results in specifications that are an order of magnitude smaller than the equivalent specifications with comparable general-purpose program synthesisers.

Search space of programs: For finite-state programs, the language in which we synthesise our programs is universal, i.e. every finite function is computed by at least one program in our language. Our solution language also has first-class support for *programs that compute multiple outputs* as well as *constants*. The former allows the direct encoding of lexicographic ranking functions of unbounded dimension [23], whereas the latter improves the efficiency when synthesising programs with non-trivial constants (as shown by our experimental results).

Search technique: An important aspect of our synthesis algorithm is how we search the space of candidate programs. We parametrise the solution language, which induces a lattice of progressively more expressive languages. As well as providing an automatic search procedure, this parametrisation increases the efficiency of our system since languages low down the lattice are easy to decide safety for.

Contributions:

- We define the synthesis fragment (Sec. 5.1.2) and show that its decision problem over finite domains is NEXPTIME-complete (Sec. 5.3.1).
- We build a program synthesis engine able to handle the use cases described in the introduction. While we focus on the synthesis of *loop-free programs over bit-vectors*, which are sufficient for most of our use cases, we also illustrate how to extend our synthesiser for generating programs with potentially unbounded loops over heap containers (cf. Chapter 6).
- We show how the synthesis fragment can be used to express several program synthesis problems, e.g., safety (Sec. 5.4.1), bug finding (Sec. 5.4.2), and refactoring (Chapter 6).
- We propose the use of second-order tautologies for avoiding unsatisfiable instances when solving program analysis problems with program synthesis (Sec. 5.5.1).
- We demonstrate our program synthesiser implementation and its performance on a set of program synthesis problems. Our experimental results show that, on benchmarks generated from static analysis, our program synthesiser compares positively with specialised tools in each area as well as with general-purpose synthesisers (Sec. 5.5).
- We present our Java Stream Theory (JST) to reason over collections and stream queries in Java. We use our program synthesiser over this logic and implement an automated refactoring decision procedure for the "Loop to Java 8 Stream" refactoring (cf. Chapter 6).

5.1 Program Analysis using the Synthesis Fragment of Second-Order Logic

5.1.1 Example problem

Program analysis problems can be reduced to the problem of finding solutions to a second-order constraint [47, 44]. In this section, we briefly discuss the constraints generated when proving safety. Note that this section gives only a brief description of the encoding of some program analyses and, later in the document, we will present the actual instantiations of our framework for all those exemplars (Sections 5.4.1 to 5.4.2).

When describing analyses that process programs with loops, we will characterise each loop by its initial state I, guard G and transition relation T.

Safety invariants: Safety checking is one of the most basic program analysis tasks. Given a safety assertion A, a safety invariant is a set of states S that is inductive with respect to the program's transition relation, and that excludes an error state. A predicate S is a safety invariant iff it satisfies the following criteria:

$$\exists S.\forall \vec{x}, \vec{x'}.I(\vec{x}) \rightarrow S(\vec{x}) \wedge \tag{5.1}$$

$$S(\vec{x}) \wedge G(\vec{x}) \wedge T(\vec{x}, \vec{x'}) \rightarrow S(\vec{x'}) \wedge \tag{5.2}$$

$$S(\vec{x}) \wedge \neg G(\vec{x}) \rightarrow A(\vec{x}) \tag{5.3}$$

Conjunct (5.1) says that each state reachable on entry to the loop is in the set S, and in combination with conjunct (5.2) shows that every state that can be reached by the loop is in S. The final conjunct (5.3) says that if the loop exits while in an S-state, the assertion A is not violated. Note that since we quantify over the set of states S, this constitutes a second-order constraint.

5.1.2 The Synthesis Fragment

We provided an example of a logical formulation of a specific static analysis problem, and now identify a logic expressive enough to encode those formulas and to extend to further, similar program analysis problems. We refer to the logic as the *synthesis fragment*[1], a fragment of second-order logic with restrictions on the use of quantification.

[1]We will discuss the relation with program synthesis in Sec. 5.2

Definition 1 (Synthesis Fragment (SF))**.** *A formula is in the* synthesis fragment *iff it is of the form*

$$\exists \vec{P}.\vec{Q}\vec{x}.\sigma(\vec{P},\vec{x})$$

where the $\vec{P}$ *range over functions, the* $\vec{Q}$ *are either* $\exists$ *or* $\forall$*, the* $\vec{x}$ *range over ground terms, and* σ *is a quantifier-free formula.*

If a pair $(\vec{P},\vec{x})$ is a satisfying model for the synthesis formula, then we write $(\vec{P},\vec{x}) \models \sigma$. For the remainder of the presentation, we drop the vector notation and write x for $\vec{x}$, with the understanding that all quantified variables range over vectors.

Expressiveness of the Synthesis Fragment: As illustrated in Sec. 5.2, finding a satisfying model for SF is an undecidable problem. It is important to stress that, as we explain in Sec. 5.3, our synthesis algorithm implementation reasons only over finite domains, rendering the problem decidable. While we use second-order logic syntax to express our constraints, the implemented algorithm solving SF is strictly less expressive than plain second-order logic.

Existential second-order syntax: The shape of SF limits it to existential second-order formulas, meaning that no quantifiers over functions are allowed apart from the initial existential quantifier. This is a limitation of the application of program synthesis that we address using our solver for SF. We illustrate in Appendix A that our solver implementation can handle more generic constraints. However, a formula of the form $\exists \vec{P_x}.\vec{Q}\vec{P_y}.\sigma(\vec{P_x},\vec{P_y})$ describes a desired program $\vec{P_x}$ by its behaviour with respect to a quantified set of other possible programs $\vec{P_y}$. For the program synthesis applications we target in this thesis, e.g., the refactoring synthesis presented in Chapter 6, such constraints are not applicable.

5.2 Solving the Synthesis Fragment using Program Synthesis

A satisfying model for a formula in SF is an assignment mapping each of the second-order variables to some function of the appropriate type and arity. We are interested in generating programs that compute these functions. For this purpose, we make use of *program synthesis*.

The synthesis problem is given in the form of a specification σ, which is a function taking a program P and input x as parameters and returning a boolean telling us

```
1: function CEGIS
2:   inputs ← ∅;
3:   while TRUE do
4:     candidate ← Synth(inputs);
5:     if candidate = UNSAT then
6:       return UNSAT;
7:     result ← Verif(candidate);
8:     if result = valid then
9:       return candidate;
10:    else
11:      inputs ← inputs ∪ result;

12: function SYNTH(inputs)
13:   (i_1, ..., i_N) ← inputs;
14:   query ← ∃P.σ(i_1, P) ∧ ... ∧ σ(i_N, P);
15:   result ← Decide(query);
16:   if result.satisfiable then
17:     return result.model;
18:   else
19:     return UNSAT;

20: function VERIF(P)
21:   query ← ∃x.¬σ(x, P);
22:   result ← Decide(query);
23:   if result.satisfiable then
24:     return result.model;
25:   else
26:     return VALID;
```

Figure 5.1: Abstract refinement algorithm

whether P did "the right thing" on input x. The synthesis problem is to determine the truth of the formula given in Definition 1.

While SF is undecidable, we can sketch the design of a solver by converting the SF satisfiability problem into an equisatisfiable synthesis problem, which we then solve with a program synthesiser. This design will be elaborated next, followed by a description of how to instantiate it for the synthesis of finite-state programs in Sec. 5.3 and for synthesising programs with unbounded loops in Chapter 6.

5.2.1 Our synthesis algorithm

We use an instantiation of the CEGIS paradigm described in Sec. 2.4 to find a program satisfying our specification. Algorithm 5.1 is divided into two procedures, SYNTH and VERIF, which interact via a finite set of test vectors INPUTS.

The SYNTH procedure tries to find an existential witness P that satisfies the partial specification,

$$\exists P. \forall x \in \text{INPUTS}. \sigma(x, P)$$

If SYNTH succeeds in finding a witness P, this witness is a candidate solution to the full synthesis formula. We pass this candidate solution to VERIF, which determines if it does satisfy the specification on all inputs by checking satisfiability of the verification formula,

$$\exists x. \neg\sigma(x, P)$$

If this formula is unsatisfiable, then the candidate solution is, in fact, a solution to the synthesis formula, and so the algorithm terminates. Otherwise, the witness x is an input on which the candidate solution fails to meet the specification. This witness x is added to the INPUTS set and the loop iterates again. It is worth noting that each iteration of the loop adds a new input to the set of inputs being used for synthesis. The full CEGIS refinement loop is described in Fig. 2.9.

5.2.2 Program generation strategies

An important aspect of our synthesis algorithm is the way we search the space of candidate programs. We employ the following strategies in parallel:

1. *Symbolic Bounded Model Checking.* A complete method for generating candidates is to use BMC [22] on a programmatic representation of the synthesis problem, as illustrated in Sec. 5.3.3.

2. *Genetic Programming and Incremental Evolution.* Genetic programming (GP) [69, 15] meta-heuristic, which we adapted and optimised for the evolution of candidate programs.

The GP option provides an adaptive way for searching through the space of programs for an individual that is "fit" in some sense. We measure the fitness of an individual by counting the number of tests in INPUTS for which it satisfies the specification.

The symbolic bounded model checking strategy explores programs in increasing size and is guaranteed to find a minimal program. This is not the case for the GP strategy, which traverses a space of programs up to a configured maximum size. Our GP algorithm is not biased towards shorter solutions, but its population is influenced by solutions discovered by the symbolic strategy. If the symbolic strategy finds a solution before the GP, then a configurable portion of the GP population is replaced by this instance to transfer its properties into the pool. This mechanism is also leveraged in the initialisation of the genetic population as the GP engine is only activated after a configurable amount of solutions are found by the symbolic engine. This extends the randomly generated GP population with candidate solutions which were suitable for the INPUTS of previous iterations.

The GP iteratively evolves the population by applying the genetic operators CROSSOVER and MUTATE. CROSSOVER combines selected existing programs into new programs, whereas MUTATE randomly changes parts of a single program. Fitter

programs are more likely to be selected and be present in future generations of the GP. We bias the CROSSOVER operation to connect solutions that work on disparate subsets of INPUTS. As an example, a solution working for inputs $1-7$ is very likely to be crossed with a solution working correctly for inputs $8-10$.

Our MUTATE operator manipulates a program by either exchanging instructions while maintaining operators, such as replacing $x+y$ with $x-y$, or replacing operands, such as $x+y$ and $x+z$. This is applied to a configurable percentage of the population at each generation. The CROSSOVER operation splits the two parent programs at a random location and recombines the resulting prefixes and suffixes into two new programs. Both the MUTATE and CROSSOVER operations are restricted to produce well-formed programs only. Fig. 5.2 provides an example of a crossover operation at a given location and Fig. 5.3 illustrates an operand as well as an operator mutation operation. When a solution consists of multiple programs (e.g., a ranking function and an invariant) the CROSSOVER algorithm only recombines programs of the same category.

$$
\begin{array}{rcl}
P_a & \rightarrow & x = y + \mathbf{z}; \quad \mathbf{y = x - 1} \\
P_b & \rightarrow & \mathbf{a = b} + c; \quad c = x + 1 \\
c(P_a, P_b) & \rightarrow & \begin{cases} x = y + c; & c = x + 1 \\ \mathbf{a = b + z;} & \mathbf{y = x - 1} \end{cases}
\end{array}
$$

Figure 5.2: GA crossover example.

$$
\begin{array}{rcl}
P_a & \rightarrow & x = y + z; \quad y = x - 1 \\
m(P_a) & \rightarrow & \begin{cases} x = \mathbf{a} + z; & y = x - 1 \\ x = y + z; & y = x * 1 \\ \ldots & \end{cases}
\end{array}
$$

Figure 5.3: GA mutate example.

Instead of generating a random population at the beginning of each subsequent iteration of the CEGIS loop, we start with the population at the end of the previous iteration. The intuition here is that this population contained many individuals that performed well on the k inputs from before, so they will probably continue to perform well on the current $k+1$ inputs. In the parlance of evolutionary programming, this is known as *incremental evolution* [42].

5.3 Synthesis for Program Variables with Bit-Vector Domains

Programming languages such as C and Java use numerical data types with finite ranges, and give semantics to the arithmetic operators using fixed-width binary encodings, otherwise known as bit-vectors [54]. We are interested in solving static analysis problems for these programming languages. For this purpose, we investigate the special case of the synthesis fragment over finite domains (Sec 5.3.1) followed by using finite-state program synthesis in order to decide it (Sec 5.3.2).

5.3.1 The synthesis fragment over finite domains

When interpreting the ground terms over a finite domain $\mathcal{D}$, the synthesis fragment is decidable, and its decision problem is NEXPTIME-complete.

Theorem 5.3.1 ($SF_{\mathcal{D}}$ is NEXPTIME-complete)**.** *For an instance of Definition 1 with n first-order variables, where the ground terms are interpreted over $\mathcal{D}$, checking the truth of the formula is NEXPTIME-complete.*

Proof. For this proof, we make use of Fagin's Theorem [33], which says that the class of sets A recognisable in time $\|A\|^k$, for some k, by a nondeterministic Turing machine is exactly the class of sets definable by existential second-order sentences.

To apply Fagin's Theorem, we must establish the size of the universe it implies. Since Definition 1 uses n $\mathcal{D}$ variables, the universe is the set of interpretations of the n variables. This set has size $|\mathcal{D}|^n$, and so by Fagin's Theorem, Definition 1 over finite domains defines exactly the class sets recognisable in $(|\mathcal{D}|^n)^k$ time by a nondeterministic Turing machine. This matches the definition of the class NEXPTIME, thereby checking the validity of an arbitrary instance of Definition 1 over $\mathcal{D}$ is NEXPTIME-complete. □

We write $SF_{\mathcal{D}}$ to denote the synthesis fragment over a finite domain $\mathcal{D}$. The finite-state synthesis problem checks the truth of the formula given in Definition 2.

Definition 2 (Finite Synthesis Formula)**.**

$$\exists P. \forall x \in \mathcal{D}. \sigma(P, x)$$

Satisfiability of $SF_{\mathcal{D}}$ can be reduced to finite-state program synthesis, as shown by Theorem 5.3.2.

Theorem 5.3.2 ($SF_{\mathcal{D}}$ is Polynomial Time Reducible to Finite Synthesis)**.** *Every instance of Definition 1, where the ground terms are interpreted over $\mathcal{D}$ is polynomial-time reducible to a finite synthesis formula (i.e., an instance of Definition 2).*

Proof. We first Skolemise the instance of Definition 1 to produce an equisatisfiable second-order sentence with the first-order part only having universal quantifiers (i.e., bring the formula into Skolem normal form). This introduces a function symbol for each first order existentially quantified variable taking linear time. Next we just existentially quantify over the Skolem functions, which takes linear time and space. The resulting formula is an instance of Definition 2. □

Corollary 5.3.3. *Finite-state program synthesis is NEXPTIME-complete.*

5.3.2 A decision procedure for $SF_{\mathcal{D}}$ based on program synthesis

The following shows how the generic construction of Sec. 5.2 can be instantiated to produce a finite-state program synthesiser. A natural choice for such a synthesiser would be to work in the logic of quantifier-free propositional formulae and to use a propositional SAT or SMT-$\mathcal{BV}$ solver as the decision procedure. However, we propose a slightly different track of using a decidable fragment of C as a "high level" logic referred to as C$^-$. The characteristic property of a C$^-$ program is that safety can be decided using a single query to a Bounded Model Checker. A C$^-$ program is a C program with the following restrictions:

(i) all loops in the program must have a constant bound,

(ii) all recursion in the program must be limited to a constant depth, and

(iii) all arrays must be statically allocated (i.e., not using `malloc`) and be of constant size.

C$^-$ programs may use nondeterministic values, assumptions, and types with arbitrary but fixed width. This effectively makes C$^-$ programs textual representation of CBMC's internal *GOTO programs* with a syntax similar to the C programming language [22].

Since each loop is bounded by a constant and each recursive function call is limited to a constant depth, a C$^-$ program necessarily terminates in $O(1)$ time. If we call the largest loop bound k, then a Bounded Model Checker with an unrolling bound of k will be a complete decision procedure for the safety of the program. For a C$^-$ program of size l and with largest loop bound k, a Bounded Model Checker will create

a SAT problem of size $O(lk)$. Conversely, a SAT problem of size s can be converted trivially into a loop-free C$^-$ program of size $O(s)$. The safety problem for C$^-$ is, therefore, NP-complete, which means it can be decided efficiently for many practical instances [32].

5.3.3 Encoding the synthesis problem

We now express the SYNTH and VERIF formulae as safety properties of C$^-$ programs as shown in Fig. 5.5.

In the SYNTH portion of the CEGIS loop, we construct a program SYNTH.C, which takes as parameters a candidate program P and test inputs. The program contains an assertion which fails iff P meets the specification for each of the inputs. Finding a new candidate program is then equivalent to checking the safety of SYNTH.C. The SYNTH program is a C$^-$ program, which means we can check its safety with Bounded Model Checking.

A candidate solution P is written in a simple RISC-like language $\mathcal{L}$, whose syntax is given in Fig. 5.4. The exact C$^-$ encoding of an $\mathcal{L}$ program is shown in Fig. 5.6. The `prog_t` structure encodes a program, which is a sequence of instructions. The parameter a is the number of arguments the program takes, and c is the number of constants in the program. The i-th instruction has opcode `ops[i]`, left operand `params[i*2]`, and right operand `params[i*2 + 1]`. An operand refers to either a program constant, a program argument or the result of a previous instruction, and its value is determined at runtime as follows:

$$val(x) = \begin{cases} x < a & \text{the } x^{\text{th}} \text{ program argument} \\ a \leq x < a + c & \texttt{consts}[x - a] \\ x \geq a + c & \text{the result of the } (x - a - c)^{\text{th}} \text{ instruction} \end{cases}$$

Since any instruction whose operands are all constants can always be eliminated (since its result is a constant), a loop-free program of minimal length will not contain any instructions with two constant operands. Therefore, the number of constants that can appear in a minimal program of length l is at most l.

A program is well-formed if no operand refers to the result of an instruction not yet computed, and if each opcode is valid. We add a well-formedness constraint of the form `params[i] < (a+c+i/2)` for each instruction, which requires a linear number of well-formedness constraints. If these constraints are satisfied, then the program is well-formed.

Integer arithmetic instructions:

```
add a b     sub a b     mul a b     div a b
neg a       mod a b     min a b     max a b
```

Bitwise logical and shift instructions:

```
and  a b     or   a b     xor a b
lshr a b     ashr a b     not a
```

Unsigned and signed comparison instructions:

```
le  a b     lt  a b     sle  a b
slt a b     eq  a b     neq  a b
```

Miscellaneous logical instructions:

```
implies a b     ite a b c
```

Floating-point arithmetic:

```
fadd a b    fsub a b    fmul a b    fdiv a b
```

Figure 5.4: The language $\mathcal{L}$.

We supply an interpreter for $\mathcal{L}$, which is written in C$^-$. The signature of this interpreter is `void exec(prog_t p, int in[N], int out[M])`, where, `out` is an output parameter.

Best encoding: A sequence of instructions (as our $\mathcal{L}$ programs) is a natural encoding of a program, but we might wonder if it is the *best* encoding for our candidate programs. We show that for a reasonable set of instruction types (i.e., valid opcodes), this encoding is optimal with respect to a property defined below. An encoding scheme E takes a function f and assigns it a name s. For a given ensemble of functions F, we are interested in the worst-case behaviour of the encoding E. In other words we are interested in the quantity

$$|E(F)| = \max\{|E(f)| \mid f \in F\}.$$

If for every encoding E' we have

$$|E(F)| \leq |E'(F)|,$$

then we say that E is an *optimal encoding* for F. Similarly, if for every encoding E' we have

$$O(|E(F)|) \subseteq O(|E'(F)|),$$

then we say that E is an *asymptotically optimal encoding* for F.

The next lemma shows that languages with ITE are universal and optimal encodings for finite functions.

```
void synth() {
  prog_t p = nondet();
  int in[N], out[M];

  assume(wellformed(p));

  in = test1;
  exec(p, in, out);
  assume(check(in, out));
  ...
  in = testN;
  exec(p, in, out);
  assume(check(in, out));

  assert(false);
}
```

```
void verif(prog_t p) {
  int in[N] = nondet();
  int out[M];

  exec(p, in, out);
  assert(check(in, out));
}
```

Figure 5.5: The SYNTH and VERIF formulae expressed as a C$^-$ program.

```
typedef BV(4) op_t;                    // An opcode
typedef BV(w) word_t;                  // An L-word
typedef BV(log2⌈c+l+a⌉) param_t;       // An operand

struct prog_t {
  op_t ops[l];           // The opcodes
  param_t params[l*2];   // The operands
  word_t consts[c];      // The program constants
}
```

Figure 5.6: The C$^-$ structure we use to encode an $\mathcal{L}$ program.

Lemma 5.3.4 (Universal and Optimal Encodings for Finite Functions)**.** *For an imperative programming language including instructions for testing equality of two values (EQ) and an if-then-else (ITE) instruction, any total function $f : \mathcal{S} \to \mathcal{S}$ can be computed by a program of size $O(|\mathcal{S}| \log |\mathcal{S}|)$ bits.*

Proof. The function f is computed by the following program:

```
t1 = EQ(x, 1)
t2 = ITE(t1, f(1), f(0))
t3 = EQ(x, 2)
t4 = ITE(t3, f(2), t2)
...
```

Each operand can be encoded in $\log_2(|\mathcal{S}| + l) = \log_2(3 \times |\mathcal{S}|)$ bits. Therefore, each instruction can be encoded in $O(\log |\mathcal{S}|)$ bits and there are $O(|\mathcal{S}|)$ instructions in the program, so the entire program can be encoded in $O(|\mathcal{S}| \log |\mathcal{S}|)$ bits. □

Lemma 5.3.5. *Any representation that can encode an arbitrary total function $f : \mathcal{S} \to \mathcal{S}$ must require at least $O(|\mathcal{S}| \log |\mathcal{S}|)$ bits to encode some functions.*

Proof. There are $|\mathcal{S}|^{|\mathcal{S}|}$ total functions $f : \mathcal{S} \to \mathcal{S}$. Therefore, by the pigeonhole principle, any encoding that can encode an arbitrary function must use at least $\log_2(|\mathcal{S}|^{|\mathcal{S}|}) = O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ bits to encode some function. □

From Lemma 5.3.4 and Lemma 5.3.5, we can conclude that *any* set of instruction types that include ITE is an asymptotically optimal function encoding for total functions with finite domains.

Theorem 5.3.6. *Furthermore, our representation for candidate programs as finite lists of instructions in SSA form is optimally concise as there is no encoding that offers a shorter representation to every function.*

Proof. From Lemma 5.3.4 and Lemma 5.3.5. □

5.3.4 Parameterising the search space

A key feature of our search algorithm that applies to all three strategies above is parametrising the solution language, which induces a lattice of progressively more expressive languages. We start by attempting to synthesise a program at the lowest point on this lattice and increasing the parameters until we reach a point at which the synthesis succeeds.

As well as giving us an automatic search procedure, this parameterisation significantly increases the efficiency of our system since languages low down the lattice are easy to decide safety for. If a program can be synthesised in a low-complexity language, then the entire procedure finishes much faster than if synthesis had been attempted in a high-complexity language.

We use the following parameters.

- *Program Length: l.* The first parameter we introduce is program length denoted by l. At each iteration, we synthesise programs of length exactly l. Starting with $l = 1$, we increment l whenever we determine that no program of length l can satisfy the specification. When we do successfully synthesise a program, we are *guaranteed that it is of minimal length* since we have previously established that no shorter program is correct.

- *Word Width: w.* A solution program runs on a virtual machine that is parametrised by the *word width*, i.e., the number of bits in each internal register and immediate constant.

- *Number of Constants: c.* By minimising the number of constants appearing in a program, we can use an efficient program encoding that speeds up the synthesis procedure substantially.

5.3.5 Adjusting the search parameters

The key to our automation approach is establishing sensible way in which to adjust the parameters of the solution language to cover all possible programs. Two important components in this search are the adjustment of parameters and the generalisation of candidate solutions, both of which are discussed in the following.

After each round of SYNTH, we may need to adjust the parameters. The logic for these adjustments is given as a tree in Fig. 5.7.

Whenever SYNTH fails, we consider which parameter caused the failure with two possibilities: either the program length l or the number of allowed constants c was too small. If $c < l$, we increment c and try another round of synthesis, but allowing ourselves an extra program constant. If $c = l$, there is no point in increasing c because no minimal $\mathcal{L}$-program has $c > l$. If it did, then there would have to be at least one instruction with two constant operands. This instruction could be removed (at the expense of adding its result as a constant), which contradicts the assumed minimality

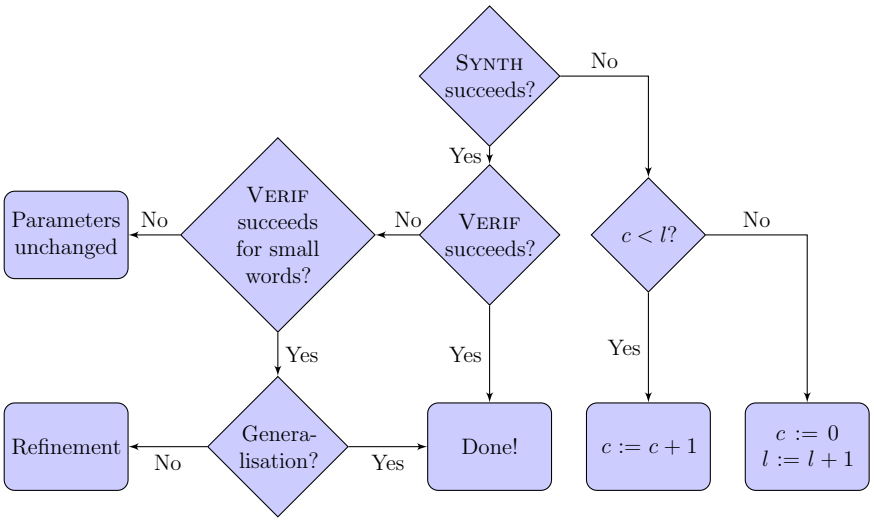


Figure 5.7: Decision tree for increasing parameters of $\mathcal{L}$.

of the program. So, if $c = l$, we set c to 0 and increment l before attempting synthesis again.

If SYNTH succeeds but VERIF fails, then we have a candidate program that is correct for some inputs and incorrect on at least one input. However, it may be the case that the candidate program is correct for *all* inputs when run on a machine with a small word size. Thus, we try to generalise the solution to a bigger word size, as explained in the next section. If the generalisation can find a correct program, then the process is complete. Otherwise, we need to increase the word width of the machine for which we are currently synthesising.

5.3.6 Generalisation of candidate solutions

It is often the case that a program satisfying the specification on a machine with $w = k$ will continue to satisfy the specification when run on a machine with $w > k$. For example, the program in Fig. 5.8 isolates the least-significant bit of a word. This is true irrespective of the word size of the machine on which it is run on, i.e., it will isolate the least-significant bit of an 8-bit word just as well as a 32-bit word. An often successful strategy is to synthesise a program for a machine with a small word size and then check if the same program is correct when run on a machine with a full-sized word.

The only drawback is that we will sometimes synthesise a program containing constants. If we have synthesised a program with $w = k$, the constants in the program will be k-bits wide. To extend the program to an n-bit machine (with $n > k$), we

```
int isolate_lsb(int x) {
  return x & -x;
}
```

Example:

x	=	1	0	1	1	1	0	1	0
-x	=	0	1	0	0	0	1	1	0
x & -x	=	0	0	0	0	0	0	1	0

Figure 5.8: A tricky bit-vector program.

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{BV}(m,m) & \rightarrow & \mathcal{BV}(n,n) & \mathcal{BV}(x,m) & \rightarrow & \mathcal{BV}(x,n) \\
\mathcal{BV}(m-1,m) & \rightarrow & \mathcal{BV}(n-1,n) & \mathcal{BV}(x,m) & \rightarrow & \mathcal{BV}(x,m)\cdot\mathcal{BV}(0,n-m) \\
\mathcal{BV}(m+1,m) & \rightarrow & \mathcal{BV}(n+1,n) & \mathcal{BV}(x,m) & \rightarrow & \underbrace{\mathcal{BV}(x,m)\cdot\ldots\cdot\mathcal{BV}(x,m)}_{\frac{n}{m}\text{times}}
\end{array}$$

Figure 5.9: Rules for extending an m-bit wide number to n-bit wide.

need to derive n-bit-wide numbers from k-bit ones. We try several strategies for this, which are given in Fig. 5.9. Here, $\mathcal{BV}(v,n)$ denotes an n-bit wide bit-vector holding the value v and $b\cdot c$ represents the concatenation of bit-vectors b and c. For example, the first rule says that if we have the 8-bit number with a value of 8 and we want to extend it to some 32-bit number, then we try the 32-bit number with a value of 32. These six rules are all heuristics that we have found to be effective in practice. The generalisations illustrated in Figs. 5.7, 5.8 and 5.9 were first introduced in Kalashnikov [32]. Our current synthesis engine expands on this with further generalisation strategies applicable to digital controller synthesis [2] (cf. Sec. 1.5.5).

5.3.7 Termination of program synthesis

For finite-state synthesis, if a specification is unsatisfiable, the algorithm still terminates with an "unsatisfiable" verdict. Intuitively, we can observe that any total function taking n bits of input is computed by some program of at most 2^n instructions. Therefore, every satisfiable specification has a solution with at most 2^n instructions. This means that if we ever need to increase the length of the candidate program we search for beyond 2^n, then we can terminate safely with knowledge that the specification is unsatisfiable.

Although this gives us a theoretical termination condition for unsatisfiable instances, a bound this high may simply be unsatisfactory in practice. To avoid such cases, we use the approach described in Sec. 5.5.1.

5.3.8 Soundness, Completeness, and Efficiency

We will now state soundness and completeness results for the $SF_{\mathcal{D}}$ solver.

Theorem 5.3.7. *Alg 5.1 is sound – if it terminates with witness P, then $P \models \sigma$.*

Proof. The procedure SYNTH terminates only if SYNTH returns "valid". In that case, $\exists x.\neg\sigma(P, x)$ is unsatisfiable and so $\forall x.\sigma(P, x)$ holds. □

Theorem 5.3.8. *Alg 5.1 with the stopping condition described in Sec 5.3.7 is complete when instantiated with $C^{\vdash}$ as a background theory – it will terminate for all specifications σ.*

Proof. Since the explicit search routine enumerates all programs (as seen by induction on the program length l), it will eventually enumerate a program that meets the specification on whatever set of inputs are currently being tracked, since by assumption such a program exists. Additionally, since safety of C^{-} programs is decidable, the query in VERIF will always provide an answer. □

According to Theorems 5.3.7 and 5.3.8, Algorithm 5.1 is sound and complete when instantiated with C^{-} as a background theory and using the stopping condition of Sec 5.3.7. This construction therefore gives as a decision procedure for $SF_{\mathcal{D}}$.

Runtime as a function of solution size: We note that the runtime of our solver is heavily influenced by the length of the shortest program satisfying the specification. If a short program exists, then the solver will find it quickly. This is particularly useful for program analysis problems where if a program exists, then most of the time many programs exist, and some are short ([65] relies on a similar remark about loop invariants).

We next show that the number of iterations of the CEGIS loop is a function of the Kolmogorov complexity of the synthesised program. The Kolmogorov complexity of a function f is defined as follows:

Definition 3 (Kolmogorov complexity)**.** *The Kolmogorov complexity $K(f)$ is the length of the shortest program that computes f.*

We can extend this definition slightly to talk about the Kolmogorov complexity of a synthesis problem in terms of its specification:

Definition 4 (Kolmogorov complexity of a synthesis problem)**.** *The Kolmogorov complexity of a program specification $K(\sigma)$ is the length of the shortest program P such that P is a witness to the satisfiability of σ.*

Let us consider the number of iterations of the CEGIS loop n required for a specification σ. Since we enumerate candidate programs in order of length, we synthesise programs with a length no greater than $K(\sigma)$ (since when we enumerate the first correct program, we will terminate). So, the space of solutions we search over is the space of functions computed by $\mathcal{L}$-programs of length no greater than $K(\sigma)$, which we will denote as the set $\mathcal{L}(K(\sigma))$. Since there are $O(2^{K(\sigma)})$ programs of length $K(\sigma)$ and some functions will be computed by more than one program, we have $|\mathcal{L}(K(\sigma))| \leq O(2^{K(\sigma)})$.

Each iteration of the CEGIS loop distinguishes at least one incorrect function from the set of correct functions, so the loop will iterate no more than $|\mathcal{L}(K(\sigma))|$ times. Therefore another bound on our runtime is $NTIME\left(2^{K(\sigma)}\right)$.

5.4 Instances of Program Synthesis Problems

The following covers the details of how to use our algorithm to solve several program analysis and synthesis problems.

5.4.1 Building a Safety Prover

To use the program synthesis based framework for constructing a safety prover, we must first look at the formulation of safety invariants (which is inside the synthesis fragment).

Safety invariants: Given a safety assertion A, a safety invariant is a set of states S that is inductive with respect to the program's transition relation, and that excludes an error state. A predicate S is a safety invariant iff it satisfies the criteria in Fig. 5.10. The first criterion is that each state reachable on entry to the loop is in the set S. The second is that every state that can be reached by the loop is in S. The final criterion says that if the loop exits while in an S-state, the assertion A is not violated.

Example 5.4.1. *The program in Fig. 5.12a is safe as x and y will not be equal regardless of how many times y gets incremented inside the loop (x is already ahead by 1). Thus, the safety invariant that our framework synthesises is $S(x, y) = x{\neq}y$.*

As we deal only with over-approximations, the generation of constraints corresponding to proving the safety of a program with nested loops is straightforward and will not be covered in this thesis. A safety prover implementation uses our program synthesis engine to find a program S that matches the requirements of a safety invariant on all program inputs.

Definition 5 (Safety Invariant [**SI**])**.**

$$\begin{aligned}\exists S.\forall x, x'.I(x) \rightarrow S(x) \wedge \\ S(x) \wedge G(x) \wedge B(x, x') \rightarrow S(x') \wedge \\ S(x) \wedge \neg G(x) \rightarrow A(x)\end{aligned}$$

Figure 5.10: Existence of a safety invariant for a single loop.

5.4.2 Building a Bug Finder

Dually to proving safety, another problem of interest is finding bugs. Ideally, if a bug exists, we would want a proof in the form of a concrete execution trace leading to it. Then, the question is how to encode the existence of such a trace in the synthesis fragment? We achieve this by introducing the notion of a *danger invariant*, which can be seen as a compact representation of an error trace [28].

The existence of a danger invariant D must show that if the loop exits having started in a D-state, an assertion will certainly fail. We require that a danger invariant is inductive with respect to the loop and that it holds in some initial state, although it need not hold in every initial state. A predicate D is a danger invariant for the loop I, G, B, A iff:

$$\exists x.I(x) \wedge D(x) \tag{5.4}$$

$$\forall x.D(x) \wedge G(x) \rightarrow \exists x'.B(x, x') \wedge D(x') \tag{5.5}$$

$$\forall x.D(x) \wedge \neg G(x) \rightarrow \neg A(x) \tag{5.6}$$

Conversely to the definition of a safety invariant where all the initial states had to be in the invariant, 5.4 says that there exists some D-state in which the loop can begin executing. Other than in the regular safety invariant use case, I represents an under-approximation to provide this guarantee. For the induction, 5.5 says that each D-state can reach at least one other D-state via an iteration of the loop. Finally, 5.6 says that if the loop exits while in a D-state, then the assertion fails.

However, this is not enough to conclude that the assertion *does* fail, since we have not yet established that the loop terminates from any D-state. So, we are in the situation where the danger invariant denotes either an assertion violation or the presence of a recurrence set. We refer to this as a *total danger invariant*.

If we want only to prove an assertion violation, then we must additionally infer a ranking function R (i.e., a function that is bounded and monotonically decreasing

with respect to the transition relation B), resulting in a *partial danger invariant* as captured in Definition 6.

Definition 6 (Partial Danger Invariant Formula [**DI**])**.**

$$\begin{aligned}\exists D, R, x_0.\forall x.\exists x'.I(x_0) \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \rightarrow B(x, x') \wedge D(x') \wedge \\ R(x) > 0 \wedge R(x) > R(x') \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x)\end{aligned}$$

Definition 7 (Skolemized Danger Invariant Formula [**SDI**])**.**

$$\begin{aligned}\exists D, R, S, x_0.\forall x.I(x_0 \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \rightarrow B(x, S(x)) \wedge D(S(x)) \wedge \\ R(x) > 0 \wedge R(x) > R(S(x)) \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x)\end{aligned}$$

Figure 5.11: Existence of a danger invariant for a single loop.

Removing the quantifier alternation: In the definition of a danger invariant, to specify that from each D-state we can reach another by iterating the loop once, we require an extra quantifier alternation. Consequently, the formula [**DI**] is not in the synthesis fragment. As our goal is to express everything in the synthesis fragment, which we can solve, we need to eliminate the extra level of quantifier alternation.

If the transition relation B is deterministic, then we do not need the quantifier alternation, since each x has precisely one successor x'. Thus, we can replace the inner $\exists x'$ in the formula [**DI**] by $\forall x'$. However, if B is non-deterministic, we must find a Skolem function which resolves the non-determinism by telling us exactly which successor is to be chosen on each iteration of the loop. This is shown in the formula [**SDI**] of Definition 7.

Example 5.4.2. *In program 5.12b, any execution trace violates the assertion unless the nondeterministic choices (denoted by "*") are such that y is incremented once less than x. One danger proof for this program is (*$(0, 1)$*, $y + 1$*, (*$x < y$*, $1000000 - x$*)) meaning that* $D(x, y) = x < y$ *holds in the initial state where* $x = 0$ *and* $y = 1$*, and it is inductive with respect to the loop's body if the nondeterministic choices are given*

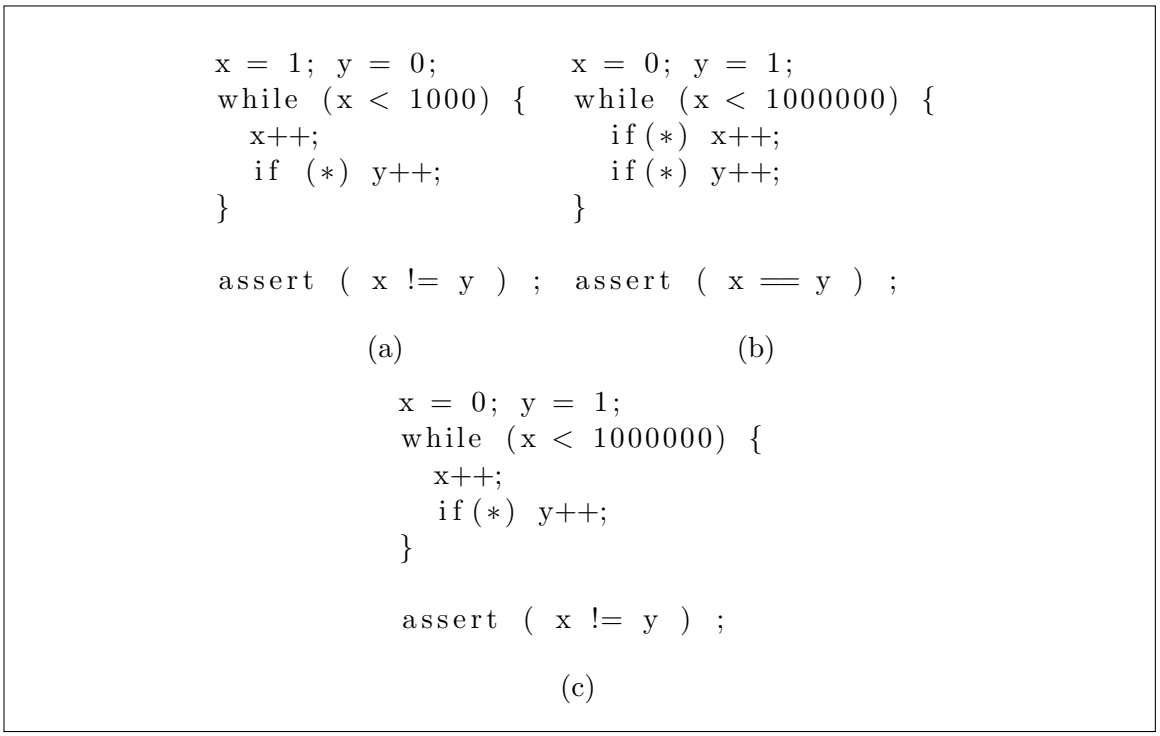


Figure 5.12: Safe and buggy examples.

by the Skolem functions $S_y(x, y) = y + 1$ *and* $S_x(x, y) = x + 1$*, respectively. In other words,*

$$\forall x, y, x'.x < y \rightarrow x' = S_x(x, y) \wedge x' < S_y(x, y)$$

The ranking function is $R(x, y) = 1000000 - x$*.*

Program 5.12c is similar to program 5.12b, with the exception that x *is incremented in each iteration and the assertion is negated. This example is more intricate as the danger invariant needs to capture the evolution of* x *and* y *from the initial state, where they are not equal, to a final state, where they are equal (and cause the assertion to fail).*

One danger proof for program 5.12c contains $D(x, y) = y = (x < 1?1 : x)$ *and* $R(x, y) = 1000000 - x$*. Essentially, this invariant says that* y *must not be incremented for the first iteration of the loop (until* x *reaches the value 1), and from this point for the rest of the iterations,* y *is incremented such that* $x = y$*. For this case,* D *is a compact and elegant representation of exactly one feasible counterexample trace. The witness Skolem function we obtain is* $S_y(x, y) = (x < 1?y : y + 1)$*.*

5.5 Implementation and Experimental Results

We implemented our decision procedure for $SF_{\mathcal{D}}$ in several tools, which we discuss in this section.

5.5.1 Avoiding Unsatisfiable Instances

As described in Sec. 5.3.8, our program synthesiser is efficient at finding satisfying assignments, when such assignments have low Kolmogorov complexity. However, if a formula is unsatisfiable, the procedure may not terminate in practice. This illustrates one of the current shortcomings of our program synthesis based decision procedure: we can only conclude that a formula is unsatisfiable once we have examined candidate solutions up to a very high length bound.

However, we note that many interesting properties of programs can be expressed as tautologies. For illustration, let us consider that we are trying to prove that a loop L is safe according to an assertion A with respect to its transition relation. Thus, following Sections 5.4.1 and 5.4.2, we can construct two formulae: one that is satisfiable iff L is safe and a second that is satisfiable iff L is unsafe. We call these formulae ϕ and ψ, respectively, and denote P_S and P_D as the proofs of safety and danger, respectively, $\exists P_S.\forall x, x'.\psi(P_S, x, x')$ and $\exists P_D.\forall x.\phi(P_D, x)$.

We can combine these as $(\exists P_S.\forall x, x'.\psi(P_S, x, x')) \vee (\exists P_D.\forall x.\phi(P_D, x))$,

which simplifies to $\exists P_S, P_D.\forall x, x', y.\, \phi(P_S, x, x') \vee \psi(P_D, y)$.

Since L either has a bug or is safe, this formula is a tautology in the synthesis fragment. Thus, either P_S or P_D must exist. In this manner, we avoid undesirable cases where we try to synthesise a solution for an unsatisfiable specification.

5.5.2 Safety and danger

To evaluate our safety and danger synthesis, we implemented the DangerZone module for the bounded model checker CBMC 5.5. We ran the resulting prover on 50 programs from the loop acceleration category in SV-COMP 2016 [93]. We selected this specific category as it has benchmarks with deep bugs and we are interested in challenging our hypothesis that danger invariants are well-suited to expose deep bugs and can complement the capabilities of existing approaches, such as BMC. Unfortunately, we had to exclude programs that make use of arrays, since these are not yet supported by the synthesiser. In addition, we introduced altered versions of the selected SV-COMP 2016 benchmarks with extended loop guards to create deeper bugs, challenging our hypothesis further. We refer to benchmarks as having a *doomed*

loop head if every state in the precondition to the successor of the loops' backwards edge implies a violation of the property. For each benchmark, we try to synthesise both a partial danger invariant (i.e., a danger invariant, a ranking function, an initial state, and Skolem functions witnessing the nondeterminism corresponding to partial correctness in Def. 7) and a total danger invariant (i.e., a danger invariant, an initial state, and Skolem functions as shown by equations 5.4, 5.5, and 5.6 in Sec. 5.4.2). To provide a comparison, we ran two state-of-the-art bounded model checking (BMC) tools, CBMC 5.5 and SMACK+CORRAL 1.5.1 [50], on the same benchmarks. In addition, we ran the benchmarks against CPAchecker 1.4 [10], the overall winner of SV-COMP 2015, and Seahorn 2.6 [48], the second-placed tool in the loops category after CPAchecker. We reproduced each tool's SV-COMP 2015 configuration with small alterations to account for the benchmarks where we increased loop guards. Finally, we manually translated the benchmarks to be compatible with Microsoft's Static Driver Verifier Research Platform (SDVRP [6]) with the Yogi 2.0 [78] back-end. Yogi's main algorithms are Synergy, Dash, Smash, and Bolt.

A benchmark contains a deep bug if it is only reachable after at least 1,000,000 unwindings. Each tool was given a time limit of 300 s, and was run on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM. The full result table of these experiments is provided in Tab. 5.1.

The results demonstrate that the DANGERZONE module outperforms all other tools on programs with deep bugs. It solves 37 (partial) and 38 (total) out of the 50 benchmarks in standalone mode, and 46 when used with CBMC. By itself, CBMC only finds 27, SMACK+CORRAL 24, CPAchecker 26, and Seahorn 31 bugs. This result is because the complexity of finding a danger invariant is orthogonal to the number of unwindings necessary to reach it. DANGERZONE's success is not determined by how deep the bug is, but by the complexity of the invariant describing it. As a result, we perform comparably on both deep and shallow bugs and can expose 18 out of the 20 deep bugs in the benchmark set. This supports our hypothesis that danger invariants are well-suited for this category of errors.

5.5.3 Controller synthesis

We implemented the tool DSSynth to use our synthesis algorithm to generate controller implementations for benchmarks selected from the literature. The first set of benchmarks uses the discrete plant of a cruise control model for a car and accounts

[2] https://github.com/diffblue/cbmc/archive/bbae05d8faecfec18a42724e72336d8f8c4e3d8d.zip

Benchmark	Deep Bugs	CBMC 5.5	SV-COMP'15 SMACK+ CORRAL 1.5.1	SV-COMP'15 CPA-checker 1.4	SV-COMP'15 Sea-horn 2.6-svn	Yogi 2.0	Dangerzone 5.5 [2] Standalone Partial	Dangerzone 5.5 [2] Standalone Total	Dangerzone 5.5 [2] with CBMC Partial	Dangerzone 5.5 [2] with CBMC Total
const1*	–	1.15 s	✗	✗	33.21 s	✗	9.09 s	0.55 s	1.15 s	0.55 s
const1t*	–	1.80 s	✗	4.01 s	0.55 s	10.09 s	5.45 s	0.64 s	1.80 s	0.64 s
const2*	–	0.36 s	3.40 s	3.54 s	0.43 s	✗	4.26 s	0.66 s	0.36 s	0.36 s
const3*†	✓	252.42 s	✗	✗	✗	✗	0.62 s	1.07 s	0.62 s	1.07 s
diamond1	–	1.13 s	22.58 s	28.25 s	0.90 s	✗	12.94 s	39.20 s	1.13 s	1.13 s
diamond1t	–	✗	✗	4.36 s	✗	9.19 s	✗	✗	✗	✗
diamond2	–	0.21 s	6.18 s	✗	0.90 s	14.46 s	✗	65.14 s	0.21 s	0.21 s
diamond2t	–	✗	✗	56.71 s	✗	✗	✗	✗	✗	✗
for1	✓	✗	✗	✗	✗	14.24 s	✗	✗	✗	✗
functions1*	✓	✗	✗	✗	✗	✗	1.36 s	1.08 s	1.36 s	1.08 s
functions1t*	✓	✗	✗	56.70 s	0.29 s	136.48 s	0.76 s	0.83 s	0.76 s	0.83 s
multivar1*	–	0.15 s	1.18 s	2.12 s	0.43 s	✗	1.23 s	0.60 s	0.15 s	0.15 s
multivar1t	–	✗	✗	1.45 s	0.30 s	10.58 s	1.53 s	1.30 s	1.53 s	1.30 s
multivar2*	–	0.18 s	1.15 s	2.11 s	0.52 s	✗	1.12 s	0.66 s	0.18 s	0.18 s
overflow1*	✓	✗	✗	✗	✗	✗	4.07 s	5.32 s	4.07 s	5.32 s
overflow1t	✓	✗	✗	58.22 s	0.27 s	✗	1.43 s	1.45 s	1.43 s	1.45 s
phases1*	✓	✗	✗	✗	✗	✗	79.41 s	3.81 s	79.41 s	3.81 s
phases1t	✓	✗	✗	58.29 s	✗	12.27 s	2.01 s	1.88 s	2.01 s	1.88 s
phases2	–	0.16 s	1.20 s	2.15 s	1.15 s	12.87 s	✗	3.67 s	0.16 s	0.16 s
phases2t	–	✗	✗	56.39 s	✗	✗	0.75 s	0.70 s	0.75 s	0.70 s
simple1*	✓	✗	✗	✗	✗	✗	7.56 s	4.36 s	7.56 s	4.36 s
simple1t	✓	✗	✗	58.31 s	0.21 s	28.12 s	1.56 s	1.52 s	1.56 s	1.52 s
simple2	–	0.15 s	1.15 s	2.13 s	1.11 s	12.52 s	8.12 s	0.88 s	0.15 s	0.15 s
simple2t	–	✗	11.55 s	1.45 s	0.21 s	11.51 s	0.51 s	0.41 s	0.51 s	0.41 s
simple3	–	0.15 s	1.12 s	2.21 s	1.03 s	✗	13.6 s	2.59 s	0.15 s	0.15 s
simple3t	–	✗	✗	57.32 s	0.22 s	✗	1.10 s	1.15 s	1.10 s	1.15 s
simple4*	✓	✗	✗	✗	✗	11.77 s	1.56 s	0.63 s	1.56 s	0.63 s
simple4t	✓	✗	✗	58.24 s	0.21 s	✗	0.50 s	0.48 s	0.50 s	0.48 s
terminator	–	0.18 s	3.02 s	✗	1.13 s	12.52 s	3.93 s	0.85 s	0.18 s	0.18 s
terminator†	✓	0.18 s	0.97 s	✗	12.48 s	1.49 s	0.98 s	0.98 s	0.18 s	0.18 s
underapprox1*	–	0.38 s	3.27 s	2.83 s	1.07 s	✗	✗	✗	0.38 s	0.38 s
underapprox1t	–	1.41 s	11.98	1.46 s	0.16 s	14.02 s	✗	✗	1.41 s	1.41 s
underapprox2*	–	0.37 s	3.08 s	2.59 s	0.84 s	✗	1.63 s	0.76 s	0.37 s	0.37 s
underapprox2t	–	1.36 s	12.39 s	1.44 s	0.16 s	12.32 s	0.76 s	0.73 s	0.76 s	0.73 s
loop1*	✓	46.59 s	✗	✗	✗	12.05 s	1.62 s	0.91 s	1.62 s	0.91 s
loop2*†	✓	✗	✗	✗	✗	✗	88.83 s	8.36 s	88.83 s	8.36 s
loop3†	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
loop4	–	0.54 s	0.15 s	✗	✗	✗	✗	✗	0.54 s	0.54 s
loop5†	✓	292.64 s	✗	✗	✗	✗	170.94 s	3.05 s	170.94 s	3.05 s
loop6	–	0.16 s	1.16 s	2.23 s	0.42 s	13.25 s	15.87 s	1.22 s	0.16 s	0.16 s
loop7	–	0.97 s	1.33 s	12.92 s	0.89 s	13.26 s	0.59 s	0.52 s	0.59 s	0.52 s
loop8†	✓	✗	✗	✗	✗	✗	2.67 s	0.83 s	1.92 s	0.96 s
loop9†	✓	✗	✗	✗	✗	✗	5.41 s	1.69 s	5.41 s	1.69 s
loop10†	✓	✗	✗	✗	✗	✗	3.86 s	1.14 s	3.86 s	1.14 s
loop11	–	0.18 s	1.15 s	2.18 s	0.42 s	12.39 s	0.48 s	0.68 s	0.48 s	0.58 s
sum01	–	0.40 s	1.23 s	✗	0.30 s	✗	✗	✗	0.40 s	0.40 s
sum01b	–	0.29 s	1.13 s	✗	0.27 s	✗	✗	✗	0.29 s	0.29 s
sum04	–	0.43 s	3.19 s	✗	0.31 s	✗	✗	✗	0.43 s	0.43 s
trex02	–	0.16 s	1.15 s	✗	0.23 s	✗	37.17 s	19.59 s	0.16 s	0.16 s
trex03	–	0.17 s	1.19 s	✗	0.27 s	10.30 s	✗	2.47 s	0.17 s	0.17 s
	Solved	28	24	26	31	21	37	40	46	46
	Avg. Time	21.57 s	4.04 s	20.75 s	2.02 s	12.89 s	13.52 s	4.60 s	8.46 s	1.11 s

Key: ✗= no result/time-out, * = contains doomed loop head, †= extended loop guard

Table 5.1: Results of safety and danger experiments.

for rolling friction, aerodynamic drag, and the gravitational disturbance force [5]. The second set of benchmarks considers a simple spring-mass damper [98]. A third set uses a physical plant for satellite applications [35]. Satellites require attitude (pose) control for orientation of antennas and sensors with respect to earth. The satellite attitude control is typically used for three-axis attitude tracking, but here we consider only one axis at a time. The final set of benchmarks considers a generic plant typically used for evaluating stability margins [59, 60].

We list the runtimes required to synthesise a stable controller for each benchmark in Table 5.2. Here, *Plant* is the discrete or continuous plant model, *Benchmark* is the name of the employed benchmark, I and F represent the number of integer and fractional bits of the stable controller, respectively, and *Gen* and *No-Gen* denote the time (in seconds) required to synthesise a stable controller for the given plant with and without generalisation (generalisation was described in Sec 5.3.6), respectively.

#	Plant	Benchmark	I	F	Gen	No-Gen
1	G_{1a}	CruiseControl02	4	16	**12 s**	67 s
2	G_{1b}	CruiseControl02†	4	16	14600 s	**52 s**
3	G_{2a}	SpgMsDamper	15	16	**52 s**	318 s
4	G_{2b}	SpgMsDamper†	15	16	✗	✗
5	G_{3a}	SatelliteB2	3	7	**36 s**	✗
6	G_{3b}	SatelliteB2†	3	7	✗	**4111 s**
7	G_{3c}	SatelliteC2	3	5	**3 s**	205 s
8	G_{3d}	SatelliteC2†	3	5	**50 s**	1315 s
9	G_4	Cruise	3	7	1 s	1 s
10	G_5	DCMotor	3	7	**1 s**	10 s
11	G_6	DCServomotor	4	11	**46 s**	✗
12	G_7	Doyleetal	4	11	**8769 s**	✗
13	G_8	Helicopter	3	7	**44 s**	✗
14	G_9	Pendulum	3	7	**1 s**	14826 s
15	G_{10}	Suspension	3	7	**1 s**	5 s
16	G_{11}	Tapedriver	3	7	1 s	1 s
17	G_{12a}	a_ST1_IMPL1	16	4	**11748 s**	✗
18	G_{12a}	a_ST1_IMPL2	16	8	**351 s**	✗
19	G_{12a}	a_ST1_IMPL3	16	12	**8772 s**	✗
20	G_{12b}	a_ST2_IMPL1	16	4	**1128 s**	✗
21	G_{12b}	a_ST2_IMPL2	16	8	✗	✗
22	G_{12b}	a_ST2_IMPL3	16	12	**15183 s**	✗
23	G_{12c}	a_ST3_IMPL1	16	4	✗	✗

Table 5.2: Results of the controller synthesis experiment.

The generalisation is based on word-width and model features. For the latter, the generalisation-based configuration abstracts away fixed point errors, which may occur in the model of the plant during the synthesis stage and only considers them during generalisation to verify whether a candidate solution holds for plants with error

models. The *No-Gen* configuration does not apply generalisation and models fixed point errors directly in the synthesis phase. For the majority of our benchmarks, the generalising configuration is much faster than the non-generalising one, with the latter timing-out in 12 out of 23 cases (with a time-out of 8 hours).

The median runtime for our benchmark set is 48 s, implying that DSSynth can synthesise half of the controllers in less than one minute. Overall, the average synthesis time amounts to approximately 42 minutes. The synthesised controllers were confirmed to be stable outside of our model representation using MATLAB. A link to the full experimental environment, including scripts to reproduce the results, all benchmarks, and the DSSynth tool, is provided in the footnote.[3]

5.5.4 Discussion of synthesis process

To help understand the role of the different solvers involved in the synthesis process, we provide a breakdown of how often each solver "won", i.e., was the first to return an answer, as outlined in Table 5.3a. We see that GP provides about 80% of the candidates, whereas CBMC provides 20%. The benchmark analysis suggests that GP progresses along the counterexample trajectory more quickly, but CBMC is very effective at pushing GP out of local minima.

CBMC	GP
19.74%	80.26%

(a) Found candidates per back-end

SYNTH	VERIF
98.40%	1.60%

(b) Runtime per phase

Table 5.3: Breakdown of successful candidate generation and runtime per phase.

Table 5.3b provides a breakdown of where the CEGIS runtime is spent with over 98% of the time in the synthesis phase, leaving less than 2% for the verification phase. This suggests that the task of verifying an existing solution is negligible when compared to generating a candidate solution satisfying a set of given counterexamples.

5.5.5 Comparison to SyGuS

To compare our CEGIS engine to other synthesis engines, we translated the 20 safety benchmarks into the SyGuS format [4] for the bit-vector theory. We then ran the following solvers:

[3] `http://www.cprover.org/DSSynth/experiment.tar.gz`

	#Solved	#TO	#Crashes	Avg. time
CBMC CEGIS	18	2	0	19.6 s
eSolver	7	5	8	13.6 s
CVC4	5	13	2	32.3 s

Table 5.4: Comparison of CBMC CEGIS engine, eSolver, and CVC4 on a subset of the safety benchmarks.

- The enumerative CEGIS solver eSolver, winner of the SyGuS 2014 competition, based on the version from the SyGuS GitHub repository on 5/7/2015.
- The program synthesiser in CVC4 by [83], winner of the SyGuS 2015 competition, based on the version for the SyGuS 2015 competition on the StarExec platform.

We could not compare against ICE-DT [40], the winner of the invariant generation category in the SyGuS 2015 competition, as it does not offer support for bit-vectors. Our comparison only uses 20 of the 96 benchmarks as we had to manually convert from our specification format (a subset of C) into the SyGuS format.

The results of these experiments are given in Table 5.4, which contains the number of benchmarks solved correctly, the number of timeouts, the number of crashes (exceptions thrown by the solver), the mean time to successfully solve, and the total number of lines in the 20 specifications.

Since the eSolver tool crashed on many of the instances, we reran the experiments on the StarExec platform to verify we had not made mistakes setting up our environment. However, the same instances also caused exceptions on StarExec.

Overall, we can see that our CEGIS engine performs better on these benchmarks than eSolver and CVC4, which validates our claim that CBMC CEGIS is suitable for program synthesis problems.

We noticed that for many cases in which eSolver and CVC4 timed out, CBMC CEGIS found a solution that involved non-trivial constants. Since the SyGuS format represents constants in unary (as chains of additions), finding programs containing constants or finding existentially quantified first order variables is expensive. Our engine's strategies for finding and generalising constants make it much more efficient at this subtask.

Chapter 6

Refactoring Synthesis

In Sec. 1.2, we illustrated the difference between *syntax*- and *semantics*-driven refactorings and highlighted the shortcomings of *syntax*-driven approaches. The introduction of our program synthesis engine in Chapter 5 provides a powerful tool to model fully automated, *semantics*-driven refactorings. There exists an expansive space of methods that can reason about program semantics. The desire to perform refactorings safely suggests the use of techniques that overapproximate program behaviours. As one possible embodiment of semantics-driven refactoring, we leverage software verification technologies with the goal of reliably automating refactoring decisions based on program semantics, as in the case of the *Substitute Algorithm* refactoring. Our research hypothesis is that semantics-driven refactorings are more precise and can handle more complex code scenarios in comparison with syntax-driven refactorings.

Demonstrator: Refactoring Iteration over Collections. We use the *Replace Loop by Java 8 Stream query*, introduced in Sec. 1.3, as a demonstrator for our idea. The refactoring falls into the *Substitute Algorithm* category as defined by Fowler [34]. To compare the original and refactored state of refactoring explicitly, we distinguish *external* from *internal* iteration.

To enable external iteration, a Collection provides the means to enumerate its elements by implementing `Iterable`. Clients using an external iterator must advance the traversal and request the next element explicitly from the iterator. External iteration has a few shortcomings:

- It is inherently sequential and must process the elements in the order specified by the collection. This bars the code from using concurrency to increase performance.

- It does not describe the intended functionality, only that each element is visited. Readers must deduce the actual semantics, such as finding an element or transforming each item, from the loop body.

The alternative to external iteration is internal iteration where, instead of controlling the iteration, the client passes an operator to be performed to an iteration procedure implemented in the `Collection` type. This procedure then applies the operation to the elements in the collection based on the algorithm the procedure implements. Examples of internal iteration patterns include finding an element by a user-provided predicate or transforming each element in a list using a provided transformer. To enable internal iteration, Java SE 8 introduces a new abstraction called *Stream* that lets users process data in a declarative way. The Stream package provides implementations of common internal iteration algorithms, such as *foreach*, *find*, and *sort* using optimised iteration orders and even concurrency where applicable. Users can leverage multi-core architectures transparently without having to write multi-threaded code. Internal iterations using Stream also explicitly declare the intended functionality through domain-specific algorithms. A call to Java 8 *find* using a predicate immediately conveys the code's intent, whereas an externally iterating `for` loop implementing the same semantics is more difficult to understand. Additionally, external iteration using a `for` loop violates Thomas' *DRY* principle (*"Don't repeat yourself"* [52]) if the intended functionality is available as a Stream template. Internal iteration through Stream thus eliminates code duplication.

For illustration, consider a further example in Fig. 6.1 (a). This example uses external iteration to create a new list by multiplying all the positive values in the list `list` by 2. In this variant of the code, we use a `while` loop to sequentially process the elements in the list.

In Fig. 6.1 (b), we re-write the code using streams, which does not use a loop statement to iterate through the list. Instead, the iteration is done internally by the stream. Essentially, we create a stream of Integer objects via `Collection.stream()`, filter it to produce a stream containing only positive values, and then transform it into a stream representing the doubled values of the filtered list.

Goal: In this section, we are interested in refactoring Java code handling collections through external iteration to use streams. Our refactoring procedure is based on the program semantics and makes use of program synthesis.

```
Iterator<Integer> it = l.iterator();
List<Integer> l2 = new ArrayList<>();
while (it.hasNext()) {
  int el = it.next().intValue();
  if (el > 0)
    newList.add(2 * el);
}
```

(a) External iteration.

```
List<Integer> newList =
        l.stream()
         .filter(el -> el>0)
         .map(el -> 2 * el);
         .collect(toList());
```

(b) Internal iteration.

Figure 6.1: Filtering and mapping examples with external (a) vs. internal (b) iteration.

```
void removeNeg(ArrayList<Integer> l) {
  Iterator<Integer> it = l.iterator();
  while (it.hasNext())
    if (it.next() < 0) it.remove();
}
List<Integer> data = new ArrayList<>();
Collections.addAll(data, 1, 2, 3);
removeNeg(data);
```

Figure 6.2: Filter example.

Contributions:

- We present a program synthesis-based refactoring procedure for Java code that handles collections through external iteration, which is a direct implementation of the refactoring introduced in Sec. 1.3.
- We present an abstraction for the Java Collection and Java Stream interfaces tailored for refactorings.
- We implement our refactoring method in the tool Kayak, and our experimental results support our conjecture that semantics-driven refactorings are more precise and can handle more complex code scenarios than syntax-driven refactorings.

General refactorings: Since our goal is to preserve generality, we are interested in refactorings that are correct independent of context. To motivate our decision, we look at the example in Fig. 6.2 where we define a method `removeNeg` that removes the negative values in the list received as an argument, which we later call for the list `data`. However, given that `data` contains only positive values, applying `removeNeg` does not have any effect.

Thus, for this particular calling context, we could refactor the body of removeNeg to a NO-OP. While this refactoring is correct for the code given in Fig. 6.2, it may cause problems during future evolution of the code, as it might eventually be used for its originally intended functionality (i.e., removing negative values). As we envision our refactoring procedure will be used during the development process, we choose not to perform such strict refactorings.

6.1 Our approach

Given an *original code* Origin, we want to infer the *refactored code* Stream such that, for any initial program state S_i, Origin and Stream produce the same final state, i.e., they are observationally equivalent. We consider a *program state* to consist of assignments to all the scalar variables plus a heap representation mapping all the Java reference variables to their corresponding heap addresses. Then,

$$\forall S_i.S_f{=}\mathrm{Origin}(S_i) \wedge S'_f{=}\mathrm{Stream}(S_i) \Rightarrow S_f{=}S'_f \tag{6.1}$$

This equivalence check can be reduced to checking the partial correctness of the triple

$$\{S_i{=}*\}\ \mathrm{Origin}\ \{\mathrm{Stream}(S_i){=}S_f\}$$

where $S_i{=}*$ means we non-deterministically select a valid initial state (i.e., we non-deterministically assign all the variables and the contents of collections). This precondition is different from *true* since it excludes heap configurations which may exist in our abstract model but are not valid in the Java runtime library. Essentially, this says that, starting with a nondeterministic state S_i, every terminating trace ends up in a state where $\mathrm{Stream}(S_i){=}S_f$ holds (we discuss non-terminating behaviours in the last paragraph of Sec. 6.4).

Note that this overapproximates the context of the initial code in the sense that it may require us to consider more initial states than those reachable at the start of Origin in the user code. As a consequence, we obtain general refactorings as explained in the previous section. In the rest of the thesis, we use the notation *equivState* to refer to the equivalence postcondition $\mathrm{Stream}(S_i){=}S_f$. Next, we explain the main steps of our refactoring procedure.

(i) Given the original code and a nondeterministic initial state as inputs, we generate the constraint $\mathit{Post}(S_i, \mathrm{Origin}) \Rightarrow \mathit{equivState}$ encoding the observational equivalence check. Here, *Post* computes the postcondition of Origin starting from the initial program state S_i. We compute *Post* by symbolically executing the code [22].

For programs with loops, we assume the existence of safety invariants and generate constraints as shown in Sec. 2.2. These safety invariants, Inv, are synthesised together with *equivState* in the next step.

(ii) We provide the equivalence constraints to our program synthesiser (see Sec. 6.4), which generates an equivalence proof in the form of *equivState* and the necessary safety invariants. As *equivState* captures the semantics of the refactored code, `Stream` can be generated directly from it. Moreover, given that *equivState* is a postcondition of the original code, the refactored code is *guaranteed* to be equivalent to the original one by construction.

Logical encoding: To generate the constraints described in (i), we must identify a logical encoding for our analysis, which we use to express *Inv* and *equivState*. As *equivState* must capture the semantics of the stream refactoring as well as equivalence between program states, our logic must have the ability to express: (1) operations supported by the Java Collection interface, (2) operations supported by the Java Stream interface, as well as (3) equality between collections (for lists, this implies we must be able to reason about both content of lists and the order of elements).

For this purpose, we define the Java Stream Theory (JST), which is informally presented in Fig. 6.3 (the formal description is provided in Sec. 6.3). For brevity, we omit some of the operations with the same semantics as their direct counterpart provided by the Java Collection or Stream API. Additionally, we use the notion of an incomplete collection/list represented by a *list segment* $x{\rightarrow}^{*}y$, i.e., the list starting at the node pointed by x and ending at the node pointed to by y.

Throughout the thesis, we take the liberty of referring to collections as *lists* (we will explain in Sec. 6.3 why, for our analysis, lists can also be used as a representation for other types of collections, e.g., sets). Also, we capture side-effects by explicitly naming the current heap – heap variables h, h' etc. are introduced (as a front end transformation), denoting the heap in which each function is to be interpreted. The mutation operators (e.g., *get*, *add*, *set*, and *remove*) then become pure functions mapping heaps to heaps.

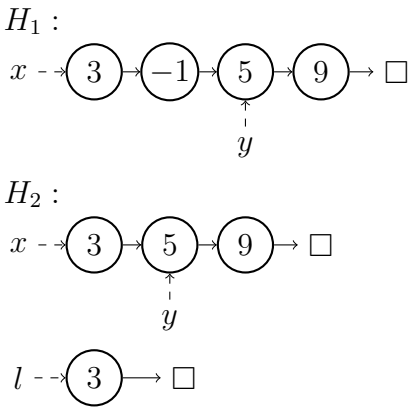
We illustrate JST through the graphical representation given in Fig. 6.4 where the circles denote the nodes in the list with their associated values. We use the dashed arrows to represent the list references. Note that heap H_2 returned by the *filter* method contains both the list received as argument and the result list l.

$alias(h, x, y)$:	do x and y point to the same node in heap h?
$size(h, x, y)$:	what is the length of the list segment from x to y in h?
$get(h, x, i)$:	what is the value stored in the i-th node of the list pointed by x in heap h?
$h' = add(h, x, i, v)$:	obtain h' from h by inserting value v at position i in the list pointed by x.
$h' = add\_last(h, x, v)$:	equivalent to $add(h, x, size(h, x, \mathbf{null}), v)$
$h' = set(h, x, i, v)$:	obtain h' from h by setting the value of the i-th element in the list pointed by x to v.
$h' = remove(h, x)$:	obtain h' from h by removing the node pointed by x. In h', x and all its aliases will point to the successor of the removed node.
$h' = removeVal(h, x, y, v)$:	obtain h' from h by removing the node with value v from the list segment $x{\rightarrow}^*y$.
$exists(h, x, y, \lambda v.P(v))$:	is there any value v in the list segment $x{\rightarrow}^*y$ such that $P(v)$ holds?
$forall(h, x, y, \lambda v.P(v))$:	is it the case that for all values $v_1 \dots v_n$ in the list segment $x{\rightarrow}^*y$, $P(v_1) \dots P(v_n)$ hold?
$h' = sorted(h, x, y, ret)$:	obtain h' from h by sorting the elements stored in the list segment $x{\rightarrow}^*y$ in the list ret (h' will contain both the list segment $x{\rightarrow}^*y$ and the list ret).
$h' = filter(h, x, y, \lambda v.P(v), ret)$:	obtain h' from h by creating a new list ret containing all the elements in the list segment $x{\rightarrow}^*y$ that match the predicate P.
$max(h, x, y)$:	what is the maximum value stored in the list segment $x{\rightarrow}^*y$?

Figure 6.3: Informal description of JST.

$min(h, x, y)$:	what is the minimum value stored in the list segment $x{\rightarrow}^*y$?
$h' = map(h, x, y, \lambda v.f(v), ret)$:	obtain h' from h by applying the mapping function f to each value in the list segment $x{\rightarrow}^*y$ and storing the result in the list pointed by ret.
$h' = skip(h, x, y, done, n, ret)$:	obtain h' by creating a new list ret containing the remaining elements of the list segment $x{\rightarrow}^*y$ after discarding the first n elements ($done$ denotes the number of elements that were already skipped).
$h' = limit(h, x, y, done, n, ret)$:	obtain h' by creating a new list ret containing the elements of the list segment $x{\rightarrow}^*y$, after its length was truncated to n ($done$ denotes the number of elements that were dropped).
$reduce(h, x, y, v, \lambda a\ b.f(a, b))$:	performs a reduction on the elements of the list segment $x{\rightarrow}^*y$ using the identity value v and the accumulation function f, and returns the reduced value.
$h' = concat(h, x, y, a, b, ret)$:	obtain h' from h by creating a new list ret containing all the elements in the list segment $x{\rightarrow}^*y$ followed by all the elements in the list segment $a{\rightarrow}^*b$.
$h' = copy(h, x, y, ret)$:	obtain h' by creating a new list ret that contains the elements of the list segment $x{\rightarrow}^*y$.
$h' = new(h, x)$	obtain h' from h by assigning x to point to **null**.
$equalLists(h, x, y, h', a, b)$	is list segment $x{\rightarrow}^*y$ in heap h equal to list segment $a{\rightarrow}^*b$ in heap h' (i.e., do they contain the same elements in the same order)?
$h' = getIterator(h, x, i, it)$	obtain heap h' by creating a new iterator it that points to the i-th element in the list pointed-to by x.

Figure 6.3: Informal description of JST.



$alias(H_1, x, y) = \text{false}$
$forall(H_1, x, y, \lambda v.v \leq 4) = \text{false}$
$exists(H_1, x, \textbf{null}, \lambda v.v = -1) = \text{true}$
$size(H_1, x, \textbf{null}) = 4$
$min(H_1, x, y) = -1$
$max(H_1, x, y) = 9$
$H_2 = filter(H_1, x, y, \lambda v.v > 0, l)$
$forall(H_2, l, \textbf{null}, \lambda v.v > 0) = \text{true}$

Figure 6.4: JST Example.

Discussion on aliasing: In the overview of our approach (Sec. 6.1), when expressing equivalence between program states, we only consider the variables (and collections) accessed by Origin (as opposed to all the live program variables). Thus, one might wonder if there are any side-effects due to aliasing that we are not considering. The answer is no, as our approach is safe for reference variables as well as the only two potential aliasing scenarios involving a reference variable p that is not directly used by Origin, which are the following:

1. p points to a collection that is modified by Origin. As the Stream refactoring is going to perform an equivalent transformation in-place, the refactoring will be transparent to p.

2. p is an iterator over a collection accessed by Origin. Then, if Origin modifies the collection, so will Stream, which results in p being invalidated in both scenarios. Contrary, if Origin does not modify the collection, neither will Stream, and p will not be affected in either case.

Next, we illustrate scenario 1 by considering again the method removeNeg in Fig. 6.2 with the following calling context, where we assume p points to some list and we create an alias p' of p:

```
ArrayList<Integer> p' = p;
removeNeg(p);
```

At first glance, a potential refactoring for removeNeg is:

```
l = l.stream().filter(el -> el>=0)
     .collect(toList());
```

However, this is incorrect when using the refactored function in the calling context mentioned above. While the list p points to is correct, the list pointed to by p' is not updated. Thus, after the call to removeNeg, p will correctly point to the filtered list, whereas p' will continue pointing to the old, unfiltered list. To avoid such situations, we perform refactorings of code that mutates collections in-place. Thus, a correct refactoring for method removeNeg is:

```
ArrayList<Integer> copy = new ArrayList<>(l);
l.clear();
copy.stream().filter(el -> el>=0)
             .forEachOrdered(l::add);
```

Here, we first create a copy *copy* of l. After performing the filtering on *copy*, we use *forEachOrdered*, provided by the Stream API, to add each element of the temporary stream back to the list pointed to by l (in the order encountered in the stream). Thus, we are not creating a new list with a new reference, but using the original one, which makes the refactoring transparent to the rest of the program, regardless of potential aliases.

6.2 Motivating Examples

In this section, we illustrate our refactoring procedure on three examples.

First example: We start with Fig. 6.1, where we create a new list by multiplying each positive value by 2 in the list list. As described, we must first introduce global heap variables that capture the side-effects. For this purpose, we use the following naming convention: the heap before executing the code (i.e. the initial heap for both the original and the refactored code) is called h_i. All the other heaps manipulated by the original program have subscript o, and those manipulated by the stream in *equivState* have subscript s.

```
Iterator<Integer> it = iterator(h_i,list);
List<Integer> newList;
h_o = new ArrayList<Integer>(h_i, newList);
while (hasNext(h_o, it)) {
  int (el, h_o) = next(h_o, it).intValue();
  if (el > 0) {
    h_o = add_last(h_o, newlist, 2 * el);}}
```

As mentioned in Sec. 6.1, *equivState* must capture the fact that the program configurations reached by executing the original code and the refactored code, respectively, are the same. We check this by verifying the following:

- the heap states reached by executing the original code and the refactored code, respectively, are equivalent (denoted as $h_o{=}h'_s$ below). We formally define heap equivalence as graph isomorphism in the next section (Def. 11). Informally, this means all the lists reachable from reference variables used by the original code (except for local variables not visible outside the original code), must be equal.

- all the scalar program variables accessed by the original code must have equal values after the execution of the original and refactored code, respectively. Again, we do not consider local variables that are not visible (e.g., `e1` in the original code in Fig. 6.1).

For this example, as there are no scalar variables handled by the code and visible outside, we only check heap equivalence. Thus, we find the following *equivState*:

$$\begin{aligned} &\mathit{equivState}(h_i, h_o, \mathit{list}) = (h_o{=}h'_s \wedge \\ &\quad h_s{=}\mathit{filter}(h_i, \mathit{list}, \mathbf{null}, \lambda v.v > 0, \mathit{list}') \wedge \\ &\quad h'_s{=}\mathit{map}(h_s, \mathit{list}', \mathbf{null}, \lambda v.2{\times}v, \mathit{list}'')) \end{aligned}$$

The above states that the heap h_o generated by the original code is equivalent to the heap h'_s generated by applying *filter* and *map* directly. Then, the safety invariant required to prove *equivState* is identical with *equivState* with the exception that it considers that the list pointed to by *list* has only been partially processed (up to the iterator *it*):

$$\begin{aligned} &\mathit{Inv}(h_i, h_o, \mathit{list}, \mathit{it}) = (h_o{=}h'_s \wedge \\ &\quad h_s{=}\mathit{filter}(h_i, \mathit{list}, \mathit{it}, \lambda v.v > 0, \mathit{list}') \wedge \\ &\quad h'_s{=}\mathit{map}(h_s, \mathit{list}', \mathbf{null}, \lambda v.2{\times}v, \mathit{list}'')) \end{aligned}$$

The invariant states that, given the iterator *it* over the list *list*, after processing the list up until *it*, both the original code and the stream postcondition generate the same heaps.

As JST directly models the Java Streams semantics, from an *equivState* postcondition we generate stream code (see Fig. 6.1 (b)).

Second example: Next, we provide a more involved example where the original code includes nested loops. For this purpose, we use the code for selection sort in Fig. 6.5 (a). First, we introduce the heap variable as shown in Fig. 6.5 (b). If Inv_{out} and Inv_{in} are the safety invariants for the outer and inner loops, respectively, then the constraints for the outer loop are there following, where we omit the inner loop as it follows directly from the equations (5.1), (5.2), and (5.3) in Sec. 5.1.1:

$$\forall h_i, h_o, l, j. \exists min. Inv_{out}(h_i, h_o, l, 0) \wedge \tag{6.2}$$
$$(Inv_{out}(h_i, h_o, l, j) \wedge j{<}(size(h_o, l){-}1) \wedge \tag{6.3}$$
$$Inv_{in}(h_i, h_o, l, size(h_o, l), j, min) \wedge \tag{6.4}$$
$$temp{=}get(h_o, l, j) \wedge h'_o{=}set(h_o, l, j, get(h_o, l, min)) \wedge \tag{6.5}$$
$$h_o{=}set(h'_o, l, min, temp)) \Rightarrow Inv_{out}(h_i, h_o, l, j{+}1) \wedge \tag{6.6}$$
$$Inv_{out}(h_i, h_o, l, j) \wedge j{\geq}(size(h_o, l){-}1) \Rightarrow equivState(h_i, h_o, l) \tag{6.7}$$

Constraint (6.2) says that the outer loop's invariant must hold in the initial state, constraints (6.3), (6.4), (6.5), and (6.6) check that Inv_{out} is re-established by the outer loop's body (by making use of Inv_{in}), whereas (6.7) asserts that the *equivState* postcondition must hold on exit from the outer loop. For this example, we find the solution

$$\begin{aligned}
&Inv_{out}(h_i, h_o, l, j) = equalLists(h'_o, l, it_{lj}, h'_s, l_s, it_{lsj}) \wedge \\
&\quad h_s = sorted(h_i, l, \mathbf{null}, l_s) \wedge \\
&\quad h'_o = getIterator(h_o, l, j, it_{lj}) \wedge \\
&\quad h'_s = getIterator(h_s, l_s, j, it_{lsj}) \wedge \\
&\quad max(h'_o, l, it_{lj}){\leq}min(h'_o, it_{lj}, \mathbf{null}) \\
&Inv_{in}(h_i, h, l, i, j, min) = (min(h''_o, it_{lj}, it_{li}){=}\min \wedge \\
&h'_o{=}getIterator(h_o, l, j, it_{lj}) \wedge h''_o{=}getIterator(h_o, l, i, it_{li})) \\
&equivState(h_i, h_o, l) = (h_o{=}h_s \wedge \\
&\quad h_s{=}sorted(h_i, l, \mathbf{null}, l))
\end{aligned}$$

The invariant of the outer loop expresses the fact that the lists sorted through external iteration and via the stream operation, respectively, are equal up until element j. As our theory JST supports iterator-based equality between lists (rather than index-based), we need to create iterators it_{ls} and it_{lsj} to the j-th element in the lists l and l_s, respectively. Additionally, the invariant of the outer loop captures the fact that the maximum element in the already sorted portion of list l is at most equal to the minimum element from the portion still to be sorted.

The inner loop's invariant captures the fact that the minimum element in the list segment between the j-th and the i-th element is min (program variable). The

```
void sorting(List<Integer> l) {
  int min, temp;
  for (int j = 0; j < l.size()-1; j++) {
    min = j;
    for (int i = j+1; i < l.size(); i++)
      if (l.get(i)<l.get(min)) min = i;


    temp = l.get(j);
    l.set(j, l.get(min));
    l.set(min, temp);}}
```

(a)

```
void sorting(List<Integer> l) {
  int min, temp;
  h_o = copyHeap(h_i);
  for (int j = 0; j < size(h_o,l)-1; j++) {
    min = j;
    for (int i = j+1; i < size(h_o,l); i++)
      if (get(h_i,l,i)<l.get(h_o,l,min)) min = i;

    temp = get(h_o,l,j);
    h_o' = set(h_o, l, j, get(h_o, l, min));
    h_o = set(h_o', l, min, temp);}}
```

(b)

Figure 6.5: Selection sort: (a) original code, (b) constraint.

postcondition *equivState* captures the equality between the final heap in the original program, h_o, and the final heap in the refactored code, h_s.

From postcondition *equivState*, we generate the following refactored code, where we modify l in-place by using a local copy.

```
List<Integer> sorting(List<Integer> l){
  List<Integer> copy = new List<>(l);
  l.clear();
  copy.stream().sorted()
              .forEachOrdered(l::add);}
```

Third example: In this example, we illustrate an aggregate refactoring as well as the importance of checking equivalence between heap states. For this purpose, we use the code below where we compute the sum of all the elements in the list pointed to by l while, at the same time, removing from the list pointed to by p some elements equal to the size of l.

```
Iterator<Integer> it = p.iterator();
int sum = 0;
for(i = 0; i<l.size(); i++) {
  sum += l.get(i);
  if(it.hasNext()) {
    it.next();
    it.remove();
}}
```

If we were only to verify that the scalar variables after executing the original and the refactored code, respectively, are equal and omit checking heap equivalence, then the following refactoring would be considered correct:

```
sum = l.stream().reduce(0, (a b)->a+b);
```

This refactoring ignores the modifications performed to list p and only computes the sum of elements in the list pointed to by l. In our case, we correctly find this refactoring to be unsound as the heap state reached after executing the original code (where p points to a modified list) is not equivalent to the one reached after executing this refactoring (where p points to the unmodified list). Instead, we find the following refactoring, where we correctly capture the mutation of p:

```
sum = l.stream().reduce(0, (a b)->a+b);
ArrayList<Integer> copy = new ArrayList<>(p);
p.clear();
copy.stream().skip(l.size())
             .forEachOrdered(p::add);
```

6.3 Java Stream Theory

We designed JST such that it meets the following criteria:

1. Express operations allowed by the Java Collection interface, operations allowed by the Java Stream interface as well as equality between collections (for lists, this implies we must be able to reason about both content of lists and the order of elements).

2. JST must be able to reason about the content and size of partially constructed lists (i.e., list segments), which are required when expressing safety invariants. For illustration, in Fig. 6.1, the safety invariant captures the fact that h_s is obtained from h_i by filtering the list segment $list{\rightarrow}^{*}it$.

3. JST must enable concise *equivState* postconditions and invariants as we use program synthesis to infer these. Thus, the smaller they are, the easier to synthesise.

To the best of our knowledge, no logic exists that meets all the criteria above. The majority of recently developed decidable heap logics are not expressive enough (fail points 1 and 2) [55, 81, 14, 71, 12, 32], whereas very expressive logics, such as FOL with transitive closure, are not concise and easily translatable to stream code (fail point 3).

While our theory is undecidable, we found it works well for our particular use case.

Semantics: We define the model used to interpret JST formulae. The set of reference variables is denoted by PV, which are those accessed in the code to be refactored (as opposed to all the reference variables in the program).

Definition 8 (Heap)**.** *A heap over reference variables PV is a tuple $H = \langle G, L_P, L_D \rangle$. G is a graph with vertices $V(G)$ and edges $E(G)$, $L_P : PV \to V(G)$ is a labelling function mapping each reference variable to a vertex of G, and $L_D : V(G) \to D$ is a labelling function associating each vertex to its data value (where D is the domain of the data values).*

Given that we are interested in heaps managed by Java Collections, we restrict the class of models to those where each vertex has outdegree 0 or 1 (i.e., we cannot have multiple edges coming out of a node). We assume that the reference variables include a special name **null**.

Function $val(h, x)$ returns the value stored in the node pointed to by x, $next(h, x)$ returns a reference to the next node after the one pointed to by x and it is defined as the unique vertex such that $(x, next(x)) \in E(h)$, and $add0(h, e, x)$ returns the heap obtained by appending element e at the beginning of the list pointed to by x. For the latter, we provide the pointwise definition:

$$
\begin{aligned}
&add0_V(h, e, x) \stackrel{\text{def}}{=} V(h) \cup \{q\} \quad \text{where } q \text{ is a fresh vertex} \\
&add0_{L_D}(h, e, x) \stackrel{\text{def}}{=} L_D(h)[q \mapsto e] \\
&add0_E(h, e, x) \stackrel{\text{def}}{=} E(h) \cup \{(q, L_P(h)(x))\}
\end{aligned}
$$

The semantics of JST is defined recursively in Fig. 6.6. Note that functions *minimum* and *maximum* return the minimum and maximum between the values received as arguments, respectively. While in Fig. 6.6 we provide the semantics for index-based operations (e.g., $set(x, y, i, v)$), we also support iterator-based ones (e.g., h'=$set(h, it, v)$ returns the heap obtained by setting the value of the node pointed to by it to v in heap h).

One important check that we must perform to prove equivalence between program states is that of heap equivalence. To define this notion, we first assign $PV_{\cap}$ to be the set of reference variables used by both the original and refactored code (excluding local variables, such as iterators used by only one of the codes). Then,

Definition 9. *Heap h and h' are equivalent, written as h=h', iff the underlying graphs reachable from $PV_{\cap}$ are isomorphic.*

Intuitively, this means that all the lists in h and h' pointed to by the same variable from $PV_{\cap}$, respectively, are equal.

Set collections: For our refactoring procedure, we use lists as the internal representation for collections denoting sets, meaning that we impose an order on the elements of sets. While we may miss some refactorings, this procedure is sound because if two collections are equal with respect to some order, then they are also equal when no order is imposed.

6.3.1 Decision procedure

Given a JST formula ϕ, we wish to determine whether ϕ is a tautology or whether there exists an assignment to the free variables in ϕ, usually representing a counterexample heap, which satisfies $\neg\phi$. The difficulty when searching for such an assignment is that heaps may be unbounded in size and there may be an unbounded number of heap models which need to be explored. We demonstrate that our encoding of JST has a small model property as introduced in [30], such that it is sufficient to look for counterexample heaps up to a fixed size. We first define abstract heaps as implemented in our decision procedure, then illustrate why this representation enjoys this property.

6.3.1.1 Abstract heaps

An abstract heap in our model is a heap where some of the nodes are collapsed into single abstract nodes. These abstract nodes do not maintain the values of the collapsed nodes, but instead, indicate whether a finite set of predicates holds for

$$val(h, x) = L_D(h)(x) \tag{6.8}$$

$$alias(h, x, y) \Leftrightarrow L_P(h)(x){=}L_P(h)(y) \tag{6.9}$$

$$\frac{i = 0}{get(h, x, i){=}val(h, x)} \tag{6.10}$$

$$\frac{i > 0}{get(h, x, i){=}get(h, next(h, x), i{-}1)} \tag{6.11}$$

$$\frac{alias(h, x, y)}{size(h, x, y) = 0} \tag{6.12}$$

$$\frac{\neg alias(h, x, y)}{size(h, x, y) = 1{+}size(h, next(h, x), y)} \tag{6.13}$$

$$\frac{alias(h, x, y)}{max(h, x, y) = -\infty} \tag{6.14}$$

$$\frac{alias(h, x, y)}{min(h, x, y) = \infty} \tag{6.15}$$

$$\frac{alias(h, x, y)}{exists(h, x, y, \lambda v.P(v)) = false} \tag{6.16}$$

$$\frac{alias(h, x, y)}{forall(h, x, y, \lambda v.P(v)) = true} \tag{6.17}$$

$$\frac{h' = copy(h, x, \mathbf{null}, l)}{add(h, x, 0, v){=}add0(h', v, l)} \tag{6.18}$$

$$\frac{h' = copy(h, next(h, x), \mathbf{null}, l)}{set(h, x, 0, v){=}add0(h', v, l)} \tag{6.19}$$

$$\frac{halias(h_1, x, y) \wedge alias(h_2, a, b)}{equalLists(h_1, x, y, h_2, a, b) = true} \tag{6.20}$$

$$\frac{alias(h_1, x, y) \wedge \neg alias(h_2, a, b)}{equalLists(h_1, x, y, h_2, a, b) = false} \tag{6.21}$$

$$\frac{\neg halias(h_1, x, y) \wedge alias(h_2, a, b)}{equalLists(h_1, x, y, h_2, a, b) = false} \tag{6.22}$$

Figure 6.6: Inference rules for Java Collection Theory.

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h) \cup \{ret \mapsto \mathbf{null}\}]}{copy(h,x,y,ret){=}h'} \tag{6.23}$$

$$\frac{\neg alias(h,x,y) \wedge h' = copy(h,next(h,x),y,ret)}{copy(h,x,y,ret){=}add0(h',val(h,x),ret)} \tag{6.24}$$

$$\frac{i > 0 \wedge h' = add(h,next(h,x),i{-}1,v)}{add(h,x,i,v){=}add0(h',val(h,x),next(h',x))} \tag{6.25}$$

$$\frac{i > 0 \wedge h' = set(h,next(h,x),i{-}1,v)}{set(h,x,i,v){=}add0(h',val(h,x),next(h',x))} \tag{6.26}$$

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{map(h,x,y,\lambda v.f(v),ret){=}h'} \tag{6.27}$$

$$\frac{\neg alias(h,x,y) \wedge h'{=}map(h,next(h,x),y,\lambda v.f(v),ret)}{map(h,x,y,\lambda v.f(v),ret) = add0(h',f(val(h,x)),ret)} \tag{6.28}$$

$$\frac{alias(h,x,y){\wedge}h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{skip(h,x,y,done,n,ret){=}h'} \tag{6.29}$$

$$\frac{\neg alias(h,x,y) \wedge n{>}0}{skip(h,x,y,done,n,ret){=}skip(h,next(h,x),y,done{+}1,n{-}1,ret)} \tag{6.30}$$

$$\frac{\neg alias(h,x,y) \wedge n{=}0 \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}L_P(h)(x)\}]}{skip(h,x,y,done,n,ret){=}h'} \tag{6.31}$$

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{filter(h,x,y,\lambda v.P(v),ret){=}h'} \tag{6.32}$$

$$\frac{\neg alias(h,x,y) \wedge \neg P(val(h,x))}{filter(h,x,y,\lambda v.P(v),ret){=}filter(h,next(h,x),y,\lambda v.P(v),ret)} \tag{6.33}$$

$$\frac{\neg alias(h,x,y) \wedge P(val(h,x)) \wedge h'{=}filter(h,next(h,x),y,\lambda v.P(v),ret)}{filter(h,x,y,\lambda v.P(v),ret) = add0(h',val(h,x),ret)} \tag{6.34}$$

$$\frac{(alias(h,x,y) \vee n{=}0) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{limit(h,x,y,done,n,ret){=}h'} \tag{6.35}$$

$$\frac{\neg alias(h,x,y) \wedge n{>}0 \wedge h'{=}limit(h,next(h,x),y,done{+}1,n{-}1,ret)}{limit(h,x,y,done,n,ret){=}add0(h',val(h,x),ret)} \tag{6.36}$$

Figure 6.6: Inference rules for Java Collection Theory.

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h) \cup \{ret \mapsto \mathbf{null}\}]}{sorted(h,x,y,ret) = h'} \tag{6.37}$$

$$\frac{\neg alias(h',x,y) \wedge h'{=}sorted(removeVal(h,x,y,min(h,x,y)),x,y,ret)}{sorted(h,x,y,ret) = add0(h',min(h,x,y),ret)} \tag{6.38}$$

$$\frac{\neg alias(h,x,y)}{max(h,x,y) = maximum(P(val(h,x)), max(h,next(h,x),y))} \tag{6.39}$$

$$\frac{\neg alias(h,x,y)}{min(h,x,y) = minimum(P(val(h,x)), min(h,next(h,x),y))} \tag{6.40}$$

$$\frac{\neg alias(h,x,y)}{exists(h,x,y,\lambda v.P(v)) \Leftrightarrow P(val(h,x)) \vee exists(h,next(h,x),y,\lambda v.P(v))} \tag{6.41}$$

$$\frac{\neg alias(h,x,y)}{forall(h,x,y,\lambda v.P(v)) \Leftrightarrow P(val(h,x)) \wedge forall(h,next(h,x),y,\lambda v.P(v))} \tag{6.42}$$

$$\frac{alias(h,x,y)}{reduce(h,x,y,v,\lambda a\ b.f(a,b)) = v} \tag{6.43}$$

$$\frac{\neg alias(h,x,y)}{reduce(h,x,y,v,\lambda a\ b.f(a,b)) = f(v, reduce(h,next(h,x),y,v,\lambda a\ b.f(a,b)))} \tag{6.44}$$

$$\frac{\neg alias(h_1,x,y) \wedge \neg alias(h_2,a,b)}{val(h_1,x) == val(h_2,a) \wedge equalLists(h_1,next(h_1,x),y,h_2,next(h_2,a),b)} \tag{6.45}$$

Figure 6.6: Inference rules for Java Collection Theory.

each collapsed node. While the abstract node data structure theoretically allows for arbitrary predicates, our refactoring use case focuses on predicates which are conducive to proving formulas over the operations listed in Fig. 6.6. Nodes which are not collapsed are referred to as concrete and retain their original information. This abstraction enables us to represent potentially unbounded heaps as finite abstract ones. Fig. 6.7 provides an example of an original heap H and a resulting abstract heap H' when collapsing all nodes not directly reachable from PV by the predicate $P(v) = v > 0$. P does not hold for all nodes collapsed into the first abstract node. Hence it is labelled with F. The second abstract node is marked with T since the only node collapsed into it satisfies P.

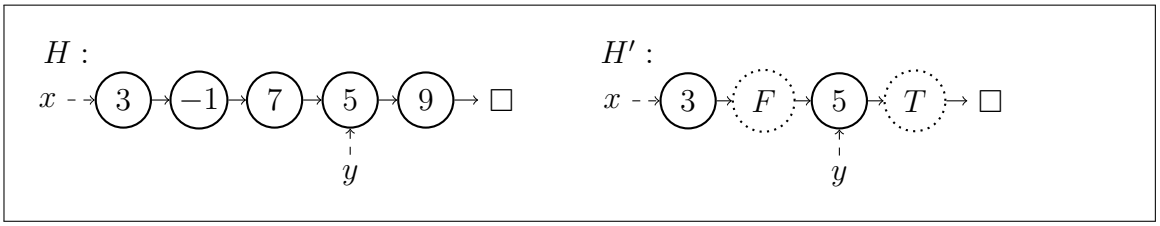


Figure 6.7: Abstract heap example.

Next, we provide a formal definition of abstract heaps in Def. 10.

Definition 10 (Abstract heap)**.** *A heap over reference variables PV is a tuple $H = \langle G, L_P, L_D, L_S \rangle$. G is a graph with concrete vertices $V_c(G)$, abstract vertices $V_a(G)$ and edges $E(G)$, $L_P : PV \to V_c(G)$ is a labelling function mapping each reference variable to a concrete vertex of G, $L_D : V_c(G) \to D$ is a labelling function associating each concrete vertex to its data value (where D is the domain of the data values), $L_S : V_a(G) \to \mathbb{N}$ is a labelling function associating each abstract vertex to its length, and $L_\forall : V_a \to \mathbb{P}$ is a labelling function mapping each abstract vertex to the set of predicates all its nodes satisfy.*

6.3.1.2 Small model property

The heaps H and H' presented in Fig. 6.7 are equivalent with respect to the observation functions in Fig .6.6. Furthermore, the transformers in JST preserve equivalence classes, as defined in Def. 11.

Definition 11 (Equivalent Heaps)**.** *Two heaps H, H' are equivalent (written $H \sim H'$) iff they are indistinguishable with respect to the JST observation functions.*

We define minimal elements of heap equivalence classes as *kernels* in Def. 12.

Definition 12 (Kernel)**.** *A kernel is a heap where every concrete node $v \in V_C(G)$ is labelled by L_P.*

Based on the definition of L_P, Def. 12 implies that a vertex v is concrete iff it is directly reachable from PV. Such a kernel heap contains at most $|PV|$ concrete nodes and at most $|PV| + 1$ abstract nodes, which represents a maximally collapsed heap. In our refactoring use case, we determine the necessary maximum heap model size by counting the transformers in the original code that access concrete value nodes.

Before stating the small model property of our logic, we define the notion of an interpretation, Γ, which is a function mapping free variables to elements of the appropriate sort. If a JST formula ϕ holds in some interpretation Γ, we say that Γ *models* ϕ and write $\Gamma \models \phi$.

Theorem 6.3.1 (JST has Small Model)**.** *For any JST formula ϕ, if there is a counterexample $\Gamma \models \neg\phi$, then there exists Γ' such that $\Gamma' \models \neg\phi$ with every heap-sorted variable in Γ' being interpreted by a kernel.*

Proof. We assume there exists a heap H such that every heap-sorted variable in Γ is interpreted by H. Then, there also exists a kernel heap H' such that $H \sim H'$. Let Γ' be an interpretation identical with Γ with the difference that the heap-sorted variables are interpreted by H'. Given that $H \sim H'$, we must have $\Gamma' \models \neg\phi$. □

6.3.2 Implementation

In CBMC, we model the Java Stream Theory explicitly as a set of transformer functions manipulating an explicit graph data structure. The model contains C structs for heaps, lists and list nodes, where we bound the maximum number of nodes per list explicitly according to Sec. 6.3.1.2. CBMC provides an API to include C models as part of its executable and load its implementations dynamically if an input program references them. The JST models and transformer functions are implemented using this API and are available in the CBMC source code at `https://github.com/diffblue/cbmc/blob/cbmc-5.7/src/ansi-c/library/jsa.h`. Fig. 6.8 illustrates the implementation of a doubly-linked list node in the JST model. Linked nodes are not identified by C pointers, but by a node identifier which represents their index in the heap's global node array. As shown in Fig. 6.9, the heap maintains an array of both concrete and abstract nodes as well as list head nodes and iterator variables.

Abstract and concrete nodes are stored in separate data structures and arrays. Each abstract node stores the length of the segment that it represents. This allows

```
typedef struct __CPROVER_jsa_concrete_node
{
  __CPROVER_jsa_node_id_t next;
  __CPROVER_jsa_node_id_t previous;
  __CPROVER_jsa_list_id_t list;
  __CPROVER_jsa_data_t value;
} __CPROVER_jsa_concrete_nodet;
```

Figure 6.8: C struct representing JST doubly-linked list nodes.

the model to represent lists of arbitrary length, positioning concrete nodes with values at arbitrary positions inside the list. Since the JST decision procedure is guaranteed only to access a limited number of concrete values in the list, this implementation is sufficient to model all required heaps for the JST logic.

```
typedef struct __CPROVER_jsa_abstract_heap
{
  // Concrete nodes store explicit element values and can be
  // positioned anywhere within a doubly-linked list.
  __CPROVER_jsa_concrete_nodet
    concrete_nodes[__CPROVER_JSA_MAX_CONCRETE_NODES];

  // One abstract list node represent multiple abstract list elements.
  // Abstract nodes do not model element values, only the number
  // elements they represent.  This allows to model lists of arbitrary
  // length with a limited number of concrete element values.
  __CPROVER_jsa_abstract_nodet
    abstract_nodes[__CPROVER_JSA_MAX_ABSTRACT_NODES];

  // All active Java iterator instances on the heap.
  __CPROVER_jsa_iteratort iterators[__CPROVER_JSA_MAX_ITERATORS];

  // Number of active iterator instances on the heap.
  __CPROVER_jsa_index_t iterator_count;

  // List head nodes, storing all active lists on the heap.
  __CPROVER_jsa_list_id_t list_head_nodes[__CPROVER_JSA_MAX_LISTS];

  // Number of active list instances on the heap.
  __CPROVER_jsa_index_t list_count;
} __CPROVER_jsa_abstract_heapt;
```

Figure 6.9: C struct for full abstract heap.

Since we eliminate loops by invariants in our decision procedure, we can determine the necessary maximum size of our heap model by counting the transformer invocations which access information from concrete nodes, such as retrieving the value from a Java

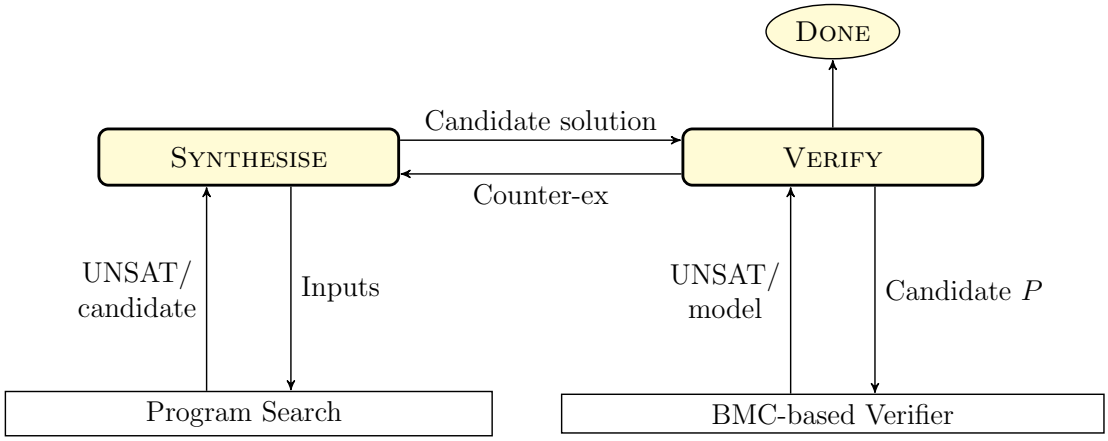


Figure 6.10: The refactoring refinement loop.

iterator variable, and bound our heap counterexamples with the necessary number of concrete nodes. For each implemented transformer matching a Java 8 Stream method, we provide a pre-configured property P for abstract nodes, which enable us to prove equivalence of the query and original loop as illustrated in Sec. 6.1. The properties used in this context establish that an abstract node is the result of a successful application of the synthesised query over the collapsed nodes. The incremental step of the proof can then be applied to the remaining concrete nodes by verifying that the original loop body transforms concrete nodes equivalently to the synthesised Java 8 Stream query.

6.4 Synthesising Refactorings

We compute the postcondition *equivState* and safety invariants by using a program synthesis engine. Such engines are used increasingly in program verification [86, 32]. Our program synthesiser makes use of Counter-Example Guided Inductive Synthesis (CEGIS) [88] for stream refactoring. We present its general architecture followed by a description of the parts specific to refactoring.

General architecture of the program synthesiser: The design of our synthesiser is illustrated in Fig. 6.10 and consists of two phases, Synthesise and Verify (cf. Sec. 5.2.1). We illustrate each of these phases by using as a running example the first motivational example in Sec. 6.2 with the goal of synthesising a solution $(equivState, Inv)$.

We start with a vacuous SYNTHESISE phase, where we generate a random candidate solution, which we pass to the VERIFY phase. For this example, let us assume that the random solution says that the stream manipulated heap is the same as the initial one (i.e., the stream code does not affect the heap), such that $equivState(h_i, h_o, list, it) = h_o{=}h'_s \wedge h'_s{=}h_s{=}h_i$.

In the VERIFY phase, we check whether the candidate solution is indeed a true solution for our synthesis problem (then we are "DONE"), or compute a counterexample. We find such a counterexample by building a program P_{verif} on which we run Bounded Model Checking [11]. BMC employs symbolic execution to map program semantics to an SAT instance [22], which verifies our equivalence constraints. If we manage to prove partial correctness of P_{verif}, then we are done. Otherwise, we provide the counterexample returned by BMC to the SYNTHESISE phase. Note that it is sound to use BMC because the program P_{verif} does not contain loops as it uses loop invariants. For the running example, BMC returns a counterexample with initial heap h'_{ce} where the candidate *equivState* is not a true postcondition when *list* contains a value 1 (added at position 0 through $add(h_{ce}, list, 0, 1)$): $h_{ce} = new(h_i, list) \ \wedge \ h'_{ce} = add(h_{ce}, list, 0, 1)$.

Next, in the SYNTHESISE phase, we add the counterexample from the previous phase to Inputs and search for a new candidate solution by constructing a program P_{synth} on which we run BMC and a genetic algorithm (GA) in parallel to find a new candidate solution that holds for all the Inputs. GA simulates an evolutionary process using selection, mutation, and crossover operators. Its fitness function is determined by the number of passed tests. GA maintains a large population of programs that are paired using crossover operation, combining successful program features into new solutions. To avoid local minima, the mutation operator replaces instructions by random values at a comparatively low probability. Moreover, we use a biased crossover operation by selecting parents that solve distinct counterexample sets for reproduction. We use the result of either BMC or GA, depending on which returns first. Again, it is sound to use BMC as the program P_{synth} does not contain loops. For the running example, BMC returns first with a candidate solution saying that the heap h'_s after the stream code is the following (for brevity, we omit the general *equivState* as, similar to before, it only denotes the fact that the original and the stream heaps are equivalent; we also omit the invariant, which is very similar to *equivState*):
$h_s = filter(h_i, list, \mathbf{null}, \lambda v.\text{true}, list') \wedge \quad h'_s = map(h_s, list, \mathbf{null}, \lambda v.2{\times}v, list'')$

This solution is almost correct, apart from the *filter* predicate, which does no actual filtering as the predicate is *true*. Returning to the VERIFY phase, we find one

further counterexample denoting a list with value 0 (which should haven been filtered out, but is not): $h_{ce}{=}new(h_i, list) \ \wedge \ h'_{ce}{=}add(h_{ce}, list, 0, 0)$.

Back in the SYNTHESISE phase, this counterexample refines the $filter$ predicate, leading to the next solution:
$h_s = \mathit{filter}(h_i, list, \mathbf{null}, \lambda v.v \neq 0, list') \wedge$
$h'_s = map(h_s, list, \mathbf{null}, \lambda v.2{\times}v, list'')$

As this still does not match the original algorithm, the VERIFY phase provides one final counterexample (a list containing value -2 that should have been filtered out, but is not):

$$h_{ce}{=}new(h_i, list) \ \wedge \ h'_{ce}{=}add(h_{ce}, list, 0, -2)$$

In the final SYNTHESISE phase, we get the solution provided in Sec. 6.2.

Elements specific to stream refactoring: To use program synthesis for stream refactoring, we required the following:

(i) *The target instruction set is JST*, which requires both the VERIFY and SYNTHESISE phases in the program synthesiser to support the JST transformers. JST directly models Java Streams such that once the synthesiser finds a solution *equivState*, we only require very light processing to generate valid Java Stream code. In particular, this processing involves the stream generation (see examples below).

Some examples of the generated stream code are provided below, where the LHS denotes either the stream heap h_s or some other scalar variable r captured by *equivState* (expressed in JST), and the RHS represents the corresponding stream refactoring. For illustration, in the first example, after the synthesiser finds that h_s in *equivState* is $h_s{=}filter(h_i, l, \mathbf{null}, \lambda v.P(v), l')$, we generate the stream refactoring by adding the stream generation $l.stream()$ before the stream filtering $filter(\lambda v.P(v))$.

Note that $\equiv$ stands for reference equality meaning that, as shown in Sec. 6.1, we must generate Java code that modifies the original collection in place.

$$h_s{=}filter(h_i, l, \mathbf{null}, \lambda v.P(v), l') \Rightarrow l' \equiv l.stream().filter(\lambda v.P(v))$$

$$h_s{=}sorted(h_i, l, \mathbf{null}, l') \Rightarrow l' \equiv l.stream().sorted()$$

$$h_s{=}skip(h_i, l, \mathbf{null}, k, 0, l') \Rightarrow l' \equiv l.stream().skip(k)$$

$$r{=}forall(h_i, l, \mathbf{null}, \lambda v.P(v)) \Rightarrow r{=}l.stream().allMatch(v \rightarrow P(v))$$

$$r{=}max(h, l, \mathbf{null}) \Rightarrow r{=}l.stream().max()$$

(ii) *The search strategy* is to parametrise the solution language, where the main parameter is the length of the solution program, denoted by l. At each iteration, we synthesise programs of length exactly l. We start with $l = 1$ and increment l whenever we determine that no program of length l can satisfy the specification. When we do successfully synthesise a program, we are *guaranteed that it is of minimal length* since we have previously established that no shorter program is correct. This is useful for our setting where we are biased towards short refactorings (see Sec. 6.6).

Terminating and exceptional behaviour: Next, we discuss how our refactoring interacts with non-terminating and exceptional behaviours of the original code.

If the original code throws an exception, then the same happens for our modelling, and we fail to find a suitable refactoring. The non-terminating behaviour can be due to either iterating over a collection with an unbounded number of elements or to a bug in the code that does not properly advance the iteration through the collection. Regarding the former, we assume the code to be refactored handles only collections with a bounded number of elements. With respect to the second reason for non-termination, if such a bug exists in the original code, then it will also exist in our modelling. Thus, we will fail to find a suitable refactoring.

6.5 Experiments

Benchmark Selection: We provide an implementation of our refactoring decision procedure, which we have named Kayak. We employed the GitHub Code Search to find relevant Java classes that contain integer collections with refactoring opportunities to streams. Kayak currently supports refactorings from Java external iterators to Streams for integer collections only. This limitation is not conceptual, but due to our Java front end based on CBMC [22], which will be extended in future work. The queries were specified conservatively as not to exceed the CBMC front end capabilities and we manually ruled out search results which cannot be implemented using the Java 8 Stream specification. We used the following search queries on 8/8/2016:

- List<Integer>+for+if+break++language%3AJava&type=Code
- List<Integer>+while+it+remove&type=Code
- List<Integer>+while+add

We found 50 code snippets with loops from the results that fit these restrictions.

Experimental Setup: To validate our hypothesis that semantics-driven refactorings are more precise than syntax-driven ones, we compare Kayak against the Integrated Development Environments IntelliJ IDEA 2016.1[1] and NetBeans 8.2[2] as well as against LambdaFicator by Franklin et al. [36]. These tools all provide a "Replace with collect" refactoring, which matches Java code against pre-configured external iteration patterns and transforms the code to a stream expression if they concur. We manually inspect each transformation for both tools to confirm correctness. Since Kayak's software synthesis can be a time-consuming process, we impose a time limit of 300 s for each benchmark. All experiments were run on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM.

Genetic algorithm configuration: We implemented a steady state genetic algorithm implementation in CEGIS with a fitness function determined by the number of passed tests. We employ a biased crossover operation, selecting parents which solve distinct counterexamples in the CEGIS counterexample set for reproduction. The intent is to combine parent refactorings that work for distinct intput sets, and thus produce offspring which behave correctly for both input sets. The population size, replacement, and mutation rates are configurable and were set to 2000, 15%, and 1%, respectively, for our experimental evaluation.

Results: Our results show that Kayak outperforms IntelliJ, NetBeans 8.2, and LambdaFicator by a significant margin. Kayak finds 39 out of 50 (78%) possible refactorings, whereas IntelliJ only transforms 10 (20%) and both NetBeans 8.2 and LambdaFicator transform 11 benchmarks (22%) successfully. IntelliJ, NetBeans, and LambdaFicator combined find 15 (30%) refactorings. This is due to the fact that there are many common Java paradigms, such as `ListIterator` or `Iterator::remove`, for which none of the tools contain pre-configured patterns and have no way of refactoring. The fact that none of the pattern-based tools account for these cases suggests that it is impractical to try to enumerate every possible refactoring pattern in IDEs.

If the pattern-based tools find a solution, then they transform the program safely and instantaneously, even in cases where Kayak fails to synthesise a refactoring within the allotted time limit. Where Kayak synthesised a valid refactoring, it did so within an average of 8.5 s. The syntax-driven tools and Kayak complement each other well

[1] https://www.jetbrains.com/idea/
[2] https://netbeans.org/

in our experiments, which is illustrated by the fact that both approaches combined would have solved 44 out of the 50 refactorings (88%) correctly. Loops matching the expected patterns of syntax-driven tools are handled with ease by such tools, regardless of semantic complexity. Kayak, on the other hand, abstracts away stark syntactical differences and recognises equivalent semantics instead, but is limited by the computational complexity of its static analysis engine.

Kayak's maximum memory usage (heap+stack) was 125MB over all benchmarks according to valgrind massif. We found that most timeouts for Kayak are due to an incomplete instruction set in the synthesis process. We plan to implement missing instructions as the program progresses beyond its research prototype phase and into an industrial refactoring tool set. We provide a link to the benchmarks used in our experiment in the footnote[3].

Benchmark	IntelliJ	NetBeans	LambdaFicator	JST
Npeople	✗	✗	✗	✓
TestStack	✗	✗	✗	✓
RemoveDuplicates	✗	✗	✗	✓
TreeSetIteratorRemoveTest	✗	✗	✗	✓
ListRemove	✗	✗	✗	✓
Sort	✗	✗	✗	✗
Chympara	✗	✗	✗	✓
SimpleArrayListTest	✗	✗	✗	✓
Esai	✗	✗	✗	✓
RemoveDuringIteration	✗	✗	✗	✓
Solution (1)	✗	✗	✗	✓
Solution (2)	✗	✗	✗	✓
ExerciseTwo	✗	✗	✗	✓
CutSticks	✗	✗	✗	✗
A_1	✗	✗	✗	✓
ExerciseThree	✓	✗	✗	✓
Solution	✓	✗	✗	✓
CollectionFilter	✗	✗	✗	✓
CutSticks (1)	✗	✓	✓	✓
CutSticks (2)	✗	✓	✓	✗
Question3_5	✗	✗	✗	✓
CutSticks (3)	✗	✗	✗	✓
CollectionTest	✗	✗	✗	✓
ListIteration	✗	✗	✗	✓
ListSetIteratorTest	✗	✗	✗	✓
TestIterator (1)	✓	✗	✗	✗
TestIterator (2)	✗	✓	✓	✓

[3] `http://www.cprover.org/refactoring/cbmc-trunk-diffblue-jst-fse-2017.tar.gz`

Benchmark	IntelliJ	NetBeans	LambdaFicator	JST
IteratorMain	✗	✗	✗	✓
FilterUneven	✗	✗	✗	✓
CheckedListBash	✓	✗	✗	✓
DataPacking	✗	✗	✗	✓
TestArrayList	✗	✗	✗	✓
ArrayUtils	✗	✓	✓	✓
GenPrime	✗	✗	✗	✗
T1E3R (1)	✗	✗	✗	✓
T1E3R (2)	✗	✗	✗	✓
Solution (3)	✗	✗	✗	✗
Euler68m	✗	✗	✗	✗
CombinationSum (1)	✓	✓	✓	✓
CombinationSum (2)	✓	✓	✓	✓
Euler2	✓	✓	✓	✗
Sets	✓	✓	✓	✓
Filter	✓	✓	✓	✗
Ex8	✗	✗	✗	✓
Test	✗	✗	✗	✓
Gray Code	✗	✗	✗	✗
Problem3	✗	✗	✗	✓
Distance	✓	✓	✓	✓
DistributedNumberOfInboundEdges	✗	✓	✓	✗
Eratosthenes	✗	✗	✗	✓
Total	10	11	11	39

Table 6.1: Results of the refactoring experiments.

6.6 Threats to Validity

We have provided exemplary evidence for our hypothesis that semantics-based refactorings can be soundly applied, are more precise, and enable more complex refactoring schemata. As we use program analysis technology, all standard threats to validity in this domain apply here as well, which we summarise briefly.

Selection of benchmarks: Our claim relates to "usual" programs written by human programmers, and our results may be skewed by the choice of benchmarks. We address this concern by collecting our benchmarks from GitHub, which hosts a representative and exceptionally large set of open-source software packages. Commercial software may have different characteristics not covered by our benchmarks, so our claim may not extend to commercial, closed-source software. Furthermore, all our benchmarks

are Java programs and our claim may not extend to other programming languages. We focused our experimental work on the exemplar of refactoring iteration over collections, and our technique may not be more widely applicable. Finally, our Java front end is still incomplete, only supporting lists of integers and lacking models for many Java system classes. This restricts our selection to a subset of the benchmarks in our GitHub search results, which may be biased in favour of our tool. We will address this issue by extending the front end to accept additional Java input.

Not supported: We exclude transformers, such as *peek* and *foreach*, which are included in the Stream API. The reason is that these transformers enable an equivalent transformation for virtually any loop processing a collection in iteration order. Fig. 6.11 illustrates such a transformation.

```
for(int e : c) { foo(e); }                (a)
c.stream().foreach(e -> { foo(e); });     (b)
```

Figure 6.11: *Foreach* example with original (a) and stream (b).

Transformations as illustrated in Fig. 6.11 do not improve expressiveness and readability of the program, and are, at best, a matter of pure *foreach* transformations. We do use them, however, to perform in-place transformations of the collections (as shown in Sec. 6.1), but they are introduced only after the actual synthesis when we generate code using streams.

Quality of refactorings: Refactorings need to generate code that remains understandable and maintainable. Syntax-driven refactoring has good control over the resulting code. The code generated by our semantic method stems from a complex search procedure, and may be challenging to read or maintain.

It is difficult to assess how well our technique does with respect to this subjective goal. First, we conjecture that small refactorings are preferable to larger ones (measured by the number of operations). Our method guarantees that we find the shortest possible refactoring due to the way we parametrise and search the space of candidate programs (as described in Sec. 6.4). It is unclear if human programmers prefer the shortest possible refactoring.

Second, our method can exclude refactorings that do not improve readability of the program. For instance, as mentioned in Sec. 6.1, we exclude transformations that include only peek and foreach, which are offered by the Stream API. A refactoring

that uses these transformers (Fig. 6.11) can be applied to virtually any loop processing a collection in iteration order but is generally undesirable.

Finally, we manually inspected the refactorings obtained with our tool and found them to represent sensible transformations.

Efficiency and scalability of the program synthesiser: We apply computationally complex program analysis techniques, implying that our broader claim is threatened by scalability limits for these techniques. The scalability of our refactoring procedure is restricted by the program synthesiser. While for most of our experiments the synthesiser could find a solution quickly, there were a few cases where it failed to find any. The problem was that the SYNTHESISE portion of the CEGIS loop failed to return with a candidate solution. Different instruction sets for the synthesis process can help mitigate this effect.

Better syntax-driven refactoring: Our hypothesis relates semantics-driven refactoring to syntax-driven refactoring. While we have undertaken every effort to identify and benchmark the existing syntax-driven refactoring methods, there may be means to achieve comparable or better results by improving syntax-driven refactoring.

Chapter 7

Conclusion

We explained in Sec. 1.2 the challenge of refactoring decisions that depend on a program's semantical properties, rather than purely syntactical features. An exemplar of such a semantics-driven refactoring was presented in Chapter 4, and we illustrated how bounded model checking can be used to establish the preconditions necessary for the respective refactoring decisions. A key characteristic of the presented refactoring was that once the preconditions were established, the actual refactoring was a well-defined code mutation. This, however, is not the case for many of the refactorings introduced in Chapter 1. Most refactoring decisions require the selection of an appropriate code mutation out of an exponential set of possible transformations.

To address this challenge, we introduced in Chapter 5 our Counterexample-Guided Inductive Synthesis engine for the construction of structured programs satisfying requested semantic properties. Several applications of this engine are presented in Sec. 5.4.1 to Sec. 5.4.2, including safety proofs and bug finding [28, 2]. Our experimental evaluation of this engine against state-of-the-art program synthesisers draws a favourable comparison for our algorithm and highlights the industrial applicability of our approach. To enable our synthesis engine to reason about programs relevant to refactorings, we introduced our Java Stream Theory JST in Sec. 6.3. JST is expressive enough to reason over the Java 8 Stream refactoring use cases described in Sec. 1.3. Based on these two individual contributions, our final research work presents a fully automated refactoring procedure for the Java 8 Stream refactoring in Sec. 6. The presented algorithm generates equivalence constraints over the JST theory and uses our synthesis engine to construct provably equivalent refactorings. When compared to industrial refactoring tools implementing the same Java 8 Stream refactoring, our algorithm implementation significantly outperformed all competitors.

Industrial applications of program synthesis algorithms are mainly limited by excessive computational complexity. However, our research highlights that our semantic

refactorings decision procedure scales well in the context of industrial benchmarks due to the abstractions inherent to the Java Stream Theory. Our research contributions thus have a palpable impact on industrial software engineering applications and have significantly advanced the state of the art in computer-assisted refactorings.

Appendix A

Example: Non-existential second order synthesis problem

Fig. A.1 provides a C representation, which can be solved by our program synthesis engine for the following second order formula: $\exists P.\forall \vec{P_{in}}, \vec{x}.\sigma(\vec{P}, \vec{x})$. Our algorithm thus synthesises a program P which for every ground term $\vec{x}$ and every combination of well-formed programs $\vec{P_{in}}$ satisfies the constraint $\sigma(\vec{P}, \vec{x})$.

```
void synth() {
  prog_t p = nondet();
  int in[N], out[M];
  prog_t in_p[N_p];

  assume(wellformed(p));

  in = test1;
  exec(p, in, out);
  assumecheck(in, in_p, out));
  ...
  in = testN;
  exec(p, in, out);
  assume(check(in, in_p, out));

  assert(false);
}
```

```
void verif(prog_t p) {
  int in[N] = nondet();
  int out[M];

  int in_p[N_p] = nondet();
  for(int i=0; i<N_p; ++i)
    assume(
      wellformed(in_p[i]));

  exec(p, in, out);
  assert(check(in, out));
}
```

Figure A.1: Second order constraint expressed as a C^- program.

While such constraints are expressible in our synthesis algorithm, none of the

applications presented in this thesis require such semantics.

Appendix B

Example: C `goto` to Java transformations

Fig. B.1 and Fig. B.2 provide extended examples of our C `goto` to Java `switch` transformations. For better readability, we extract each label section into a separate function, as illustrated in Fig. B.1. C *goto* statements further allow jumps into other control structures, such as loops or *if* statements. This does not hold for Java *switch-case* statements. Wherever Jitsune observes this behaviour, it translates the respective control structure into a series of *goto* statements and labels, which is a feature provided by CBMC [22]. Fig. B.2 illustrates this approach.

```
void jmp() {
 init:
 int i = 0;
 goto middle;

 start:
 i = 2;
 goto end;

 middle:
 i = 1;
 goto start;

 end:
 i = 3;
}
```

(a) Native JNI/C.

```
class jmp_container{
 int i;

 private int init(){
  i=0;
  return 2; // goto middle;
 }
 private int start(){
  i=2;
  return 3; // goto end;
 }
 private int middle(){
  i=1;
  return 1; // goto start;
 }
 private int end(){
  i=3
  return -1; // exit function
 }

 private int run_label(int _label){
  switch(_label){
  case 0:  return init();
  case 1:  return start();
  case 2:  return middle();
  case 3:  return end();
  default: return -1;
  }
 }

 // Function entry point
 public void run(){
  int _label=0;
  while (_label >= 0)
     _label = run_label(_label);
 }
}
```

(b) Java

Figure B.1: *Goto* translation to Java.

```
void jmp() {
  int i = 0;
  goto first;
  for(; i < 10; ++i) {
    printf(",");
    first:
    printf("%d", i);
  }
}
```

(a) Native JNI/C.

```
class jmp_container{
 int i;

 private int init(){
  i = 0;
  return 1;
 }
 private int loop_guard(){
  if(i < 10) { return 3; }
  return -1;
 }
 private int body_entry(){
  Stdio.printf(",");
  return 1;
 }
 private int first(){
  Stdio.printf("%d", i);
  i=i+1;
  return 2;
 }

 private int run_label(int lbl){
  switch(lbl){
  case 0:  return init();
  case 1:  return first();
  case 2:  return loop_guard();
  case 3:  return body_entry();
  default: return -1;
  }
 }

 // Function entry point
 public void run(){
  int _label=0;
  while (_label >= 0)
    _label = run_label(_label);
 }
}
```

(b) Java.

Figure B.2: *Goto* into control structure and translation to Java.

Appendix C

C to Jitsune expression translations

This section lists all categories of translations performed by Jitsune, starting from CBMC's internal type representation, such as `__CPROVER_bitvector[n]` for an integer bit-vector of length n [22].

```
__CPROVER_bitvector[16] x; → short x;                                    (C.1)
__CPROVER_bitvector[32] x; → int x;                                      (C.2)
__CPROVER_bitvector[64] x; → long x;                                     (C.3)
__CPROVER_floatbv[32][23] x; → float x;                                  (C.4)
__CPROVER_floatbv[64][52] x; → double x;                                 (C.5)
```

Figure C.1: Primitive type declarations.

```
__CPROVER_bitvector[16] x; →
    IntegerPointee<Short> x=new IntegerPointee<>();                      (C.6)
__CPROVER_bitvector[16] *p = &x; →
 Pointer<IntegerPointee<Short>> p=Pointers.reference(x);                 (C.7)
```

Figure C.2: Type declarations with pointer access.

```
malloc(s) → Stdlib.malloc(s)                                             (C.8)
free(p) → Stdlib.free(p)                                                 (C.9)
printf(s, o) → Stdio.printf(s, o)                                        (C.10)
```

Figure C.3: Standard library calls.

$$x + y \rightarrow \text{Operators.plus}(x, y) \tag{C.11}$$

$$x - y \rightarrow \text{Operators.minus}(x, y) \tag{C.12}$$

$$x * y \rightarrow \text{Operators.multiply}(x, y) \tag{C.13}$$

$$x / y \rightarrow \text{Operators.divide}(x, y) \tag{C.14}$$

$$++x \rightarrow \text{Operators.preIncrement}(x) \tag{C.15}$$

$$--x \rightarrow \text{Operators.preDecrement}(x) \tag{C.16}$$

$$x++ \rightarrow \text{Operators.postIncrement}(x) \tag{C.17}$$

$$x-- \rightarrow \text{Operators.postDecrement}(x) \tag{C.18}$$

Figure C.4: Operators on wrapped types.

$$\&x \rightarrow \text{Pointers.reference}(x) \tag{C.19}$$

$$*p \rightarrow \text{Pointers.dereference}(p) \tag{C.20}$$

Figure C.5: Pointer arithmetic.

$$\text{if}(c)\ \{\ \} \rightarrow \text{if}(c)\ \{\ \} \tag{C.21}$$

$$\text{if}(c)\ \{\ \}\ \text{else}\ \{\ \} \rightarrow \text{if}(c)\ \{\ \}\ \text{else}\ \{\ \} \tag{C.22}$$

Figure C.6: Branches.

```
while(c) { } → while(c) { }                    (C.23)
do { } while(c); → do { } while(c);            (C.24)
for(i; c; s) { } → for(i; c; s) { }            (C.25)
```

Figure C.7: Loops.

Appendix D

Jitsune instructions

Our Jitsune tool is available for download at `http://www.cprover.org/refactoring/jitsune.html`. The program is intended to be compiled and run on a Linux-based system with a *bash* shell. After downloading, the following instructions will execute the standard Jitsune demo benchmarks:

```
tar -xzf jitsune.tar.gz
./build.sh
./run-demos.sh
```

Appendix E

Kayak instructions

The Kayak executable and other resources are available for download at `http://www.cprover.org/refactoring/kayak.html`. On a Linux-based system, the program can be compiled and executed for the pre-configured benchmarks using the following commands:

```
tar -xzf cbmc-trunk-diffblue-jst-fse-2017.tar.gz
export PATH=<jitsune_dir>/src/cbmc:<jitsune_dir>/src/cegis:${PATH}
cd <jitsune_dir>/regression
./test.pl -c cegis cegis/cegis_jsa_benchmark_*
```

Bibliography

[1] BS ISO/IEC 9899:2011 - Information technology. Programming languages. C (2012)

[2] Abate, A., Bessa, I., Cattaruzza, D., Cordeiro, L., David, C., Kesseli, P., Kroening, D.: Sound and automated synthesis of digital stabilizing controllers for continuous plants. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC '17, pp. 197–206. ACM, New York, NY, USA (2017). DOI 10.1145/3049797.3049802. URL http://doi.acm.org/10.1145/3049797.3049802

[3] Abate, A., Bessa, I., Cattaruzza, D., Cordeiro, L., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Automated Formal Synthesis of Digital Controllers for State-Space Physical Plants, pp. 462–482. Springer International Publishing, Cham (2017). DOI 10.1007/978-3-319-63387-9_23. URL https://doi.org/10.1007/978-3-319-63387-9_23

[4] Alur, R., et al.: Syntax-guided synthesis. In: FMCAD (2013)

[5] Astrom, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press, Princeton, NJ, USA (2008)

[6] Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, chap. The Static Driver Verifier Research Platform, pp. 119–122. Springer, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-14295-6_11. URL http://dx.doi.org/10.1007/978-3-642-14295-6_11

[7] Bavota, G., De Lucia, A., Oliveto, R.: Identifying extract class refactoring opportunities using structural and semantic cohesion measures. J. Syst. Softw. **84**(3), 397–414 (2011)

[8] Beyene, T.A., Brockschmidt, M., Rybalchenko, A.: Ctl+fo verification as constraint solving. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014, pp. 101–104. ACM, New York, NY, USA (2014). DOI 10.1145/2632362.2632364. URL http://doi.acm.org/10.1145/2632362.2632364

[9] Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: CAV, LNCS, pp. 869–882. Springer (2013)

[10] Beyer, D., Keremoglu, M.: CPAchecker: A tool for configurable software verification. In: G. Gopalakrishnan, S. Qadeer (eds.) Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 6806, pp. 184–190. Springer (2011). DOI 10.1007/978-3-642-22110-1_16. URL http://dx.doi.org/10.1007/978-3-642-22110-1_16

[11] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003)

[12] Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Automated Technology for Verification and Analysis (ATVA), LNCS. Springer (2012)

[13] Brain, M., Crick, T., De Vos, M., Fitch, J.: TOAST: Applying Answer Set Programming to Superoptimisation, pp. 270–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). DOI 10.1007/11799573_21. URL http://dx.doi.org/10.1007/11799573_21

[14] Brain, M., David, C., Kroening, D., Schrammel, P.: Model and proof generation for heap-manipulating programs. In: European Symposium on Programming (ESOP), LNCS, pp. 432–452. Springer (2014)

[15] Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer (2007)

[16] Chapman, M., Chockler, H., Kesseli, P., Kroening, D., Strichman, O., Tautschnig, M.: Learning the Language of Error, pp. 114–130. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-24953-7_9. URL http://dx.doi.org/10.1007/978-3-319-24953-7_9

[17] Cheung, A., Solar-Lezama, A., Madden, S.: Optimizing database-backed applications with query synthesis. In: Conference on Programming Language Design and Implementation, PLDI, pp. 3–14 (2013)

[18] Christopoulou, A., Giakoumakis, E., Zafeiris, V.E., Vasiliki, S.: Automated refactoring to the strategy design pattern. Information and Software Technology **54**(11), 1202 – 1214 (2012)

[19] Church, A.: Logic, arithmetic, automata. In: Proc. Internat. Congr. Mathematicians, pp. 23–35. Inst. Mittag-Leffler, Djursholm (1962)

[20] Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1), 7–34 (2001)

[21] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. pp. 1512–1542 (1994)

[22] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS, *LNCS*, vol. 2988, pp. 168–176. SPRINGER (2004)

[23] Cook, B., See, A., Zuleger, F.: Ramsey vs. Lexicographic Termination Proving, pp. 47–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-36742-7_4. URL `http://dx.doi.org/10.1007/978-3-642-36742-7_4`

[24] Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A.: Source transformation in software engineering using the TXL transformation system. Information and Software Technology **44**(13), 827 – 837 (2002)

[25] Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: R. Giacobazzi, J. Berdine, I. Mastroeni (eds.) Verification, Model Checking, and Abstract Interpretation, *Lecture Notes in Computer Science*, vol. 7737, pp. 128–148. Springer (2013). DOI 10.1007/978-3-642-35873-9_10. URL `http://dx.doi.org/10.1007/978-3-642-35873-9_10`

[26] Cristina, D., Pascal, K., Matt, L., Daniel, K.: Program synthesis for program analysis. ACM Transactions on Programming Languages and Systems p. to appear (2018)

[27] David, C., Kesseli, P., Kroening, D.: Kayak: Safe semantic refactoring to java streams. Technical Report (2016)

[28] David, C., Kesseli, P., Kroening, D., Lewis, M.: Danger Invariants, pp. 182–198. Springer International Publishing, Cham (2016).

DOI 10.1007/978-3-319-48989-6_12. URL http://dx.doi.org/10.1007/978-3-319-48989-6_12

[29] David, C., Kroening, D., Lewis, M.: Danger invariants. CoRR **abs/1503.05445** (2015). URL http://arxiv.org/abs/1503.05445

[30] David, C., Kroening, D., Lewis, M.: Propositional reasoning about safety and termination of heap-manipulating programs. In: J. Vitek (ed.) Programming Languages and Systems, pp. 661–684. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[31] David, C., Kroening, D., Lewis, M.: Unrestricted termination and non-termination proofs for bit-vector programs. In: ESOP (2015)

[32] David, C., Kroening, D., Lewis, M.: Using program synthesis for program analysis. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20), LNCS, pp. 483–498. Springer (2015)

[33] Fagin, R.: Generalized first-order spectra, and polynomial. time recognizable sets. SIAM-AMS Proceedings **7**, 43–73 (1974). URL http://www.researchgate.net/publication/242608657_Generalized_first-order_spectra_and_polynomial._time_recognizable_sets

[34] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

[35] Franklin, G.F., Powell, D.J., Emami-Naeini, A.: Feedback Control of Dynamic Systems, 4th edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)

[36] Franklin, L., Gyori, A., Lahoda, J., Dig, D.: LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp. 1287–1290 (2013). DOI 10.1109/ICSE.2013.6606699. URL http://dx.doi.org/10.1109/ICSE.2013.6606699

[37] Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pp. 416–419. ACM (2011). DOI 10.1145/2025113.2025179. URL http://doi.acm.org/10.1145/2025113.2025179

[38] Fuhrer, R.M., Tip, F., Kiezun, A., Dolby, J., Keller, M.: Efficiently refactoring java applications to use generic libraries. In: ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings, pp. 71–96 (2005). DOI 10.1007/11531142_4. URL http://dx.doi.org/10.1007/11531142_4

[39] Galeotti, J.P., Fraser, G., Arcuri, A.: Extending a search-based test generator with adaptive dynamic symbolic execution. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 421–424. ACM (2014)

[40] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A Robust Framework for Learning Invariants, pp. 69–87. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-08867-9_5. URL http://dx.doi.org/10.1007/978-3-319-08867-9_5

[41] Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. In: Time for Verification, Essays in Memory of Amir Pnueli, pp. 167–184 (2010)

[42] Gomez, F., Mikkulainen, R.: Incremental evolution of complex general behavior. Adapt. Behav. **5**(3-4), 317–342 (1997). DOI 10.1177/105971239700500305. URL http://dx.doi.org/10.1177/105971239700500305

[43] Gopan, D., Reps, T.: Low-level library analysis and summarization. In: W. Damm, H. Hermanns (eds.) Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 4590, pp. 68–81. Springer (2007). DOI 10.1007/978-3-540-73368-3_10. URL http://dx.doi.org/10.1007/978-3-540-73368-3_10

[44] Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416 (2012)

[45] Gulwani, S.: Dimensions in program synthesis. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP '10, pp. 13–24. ACM, New York, NY, USA (2010). DOI 10.1145/1836089.1836091. URL http://doi.acm.org/10.1145/1836089.1836091

[46] Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI, pp. 62–73 (2011)

[47] Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. SIGPLAN Not. **43**(6), 281–292 (2008). DOI 10.1145/1379022.1375616. URL http://doi.acm.org/10.1145/1379022.1375616

[48] Gurfinkel, A., Kahsai, T., Navas, J.: SeaHorn: A framework for verifying C programs (competition contribution). In: C. Baier, C. Tinelli (eds.) Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 9035, pp. 447–450. Springer (2015). DOI 10.1007/978-3-662-46681-0_41. URL http://dx.doi.org/10.1007/978-3-662-46681-0_41

[49] Gyori, A., Franklin, L., Dig, D., Lahoda, J.: Crossing the gap from imperative to functional programming through refactoring. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 543–553. ACM, New York, NY, USA (2013). DOI 10.1145/2491411.2491461. URL http://doi.acm.org/10.1145/2491411.2491461

[50] Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+Corral: A modular verifier (competition contribution). In: C. Baier, C. Tinelli (eds.) Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 9035, pp. 450–453. Springer (2015)

[51] Hofferek, G., Gupta, A., Könighofer, B., Jiang, J.H.R., Bloem, R.: Synthesizing multiple boolean functions using interpolation on a single proof. CoRR **abs/1308.4767** (2013)

[52] Hunt, A., Thomas, D.: The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[53] Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: P. Jalote, L.C. Briand, A. van der Hoek (eds.) 36th International Conference on Software Engineering (ICSE), pp. 435–445. ACM (2014)

[54] International Standards Organization, International Electrotechnical Commission: ISO/IEC 9899:2011, Programming Languages – C, first edn. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036 (2011). URL http://www.open-std.org/jtc1/sc22/wg14/

[55] Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Computer Aided Verification (CAV), LNCS, pp. 756–772. Springer (2013)

[56] Iu, M., Cecchet, E., Zwaenepoel, W.: JReq: Database queries in imperative languages. In: Compiler Construction (CC), pp. 84–103 (2010)

[57] Jeon, S.U., Lee, J.S., Bae, D.H.: An automated refactoring approach to design pattern-based program transformations in Java programs. In: Asia-Pacific Software Engineering Conference (APSEC), pp. 337–345 (2002)

[58] Kataoka, Y., Notkin, D., Ernst, M.D., Griswold, W.G.: Automated support for program refactoring using invariants. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), ICSM '01. IEEE Computer Society (2001)

[59] Keel, L., Bhattacharyya, S.: Robust, fragile, or optimal? IEEE Trans. on Automatic Control **42**(8), 1098–1105 (1997). DOI 10.1109/9.618239

[60] Keel, L., Bhattacharyya, S.: Stability margins and digital implementation of controllers. In: Proc. American Control Conference, vol. 5, pp. 2852–2856 (1998). DOI 10.1109/ACC.1998.688377

[61] Kerievsky, J.: Refactoring to patterns. In: Extreme Programming and Agile Methods, *LNCS*, vol. 3134, p. 232. Springer (2004)

[62] Khatchadourian, R., Sawin, J., Rountev, A.: Automated refactoring of legacy java software to enumerated types. In: Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, pp. 224–233 (2007)

[63] Kiezun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.: Refactoring for parameterizing java classes. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, pp. 437–446 (2007). DOI 10.1109/ICSE.2007.70. URL `http://dx.doi.org/10.1109/ICSE.2007.70`

[64] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). DOI 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`

[65] Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates, pp. 328–343. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-17164-2_23. URL http://dx.doi.org/10.1007/978-3-642-17164-2_23

[66] Kraan, I., Basin, D., Bundy, A.: Logic program synthesis via proof planning. In: Logic Program Synthesis and Transformation, pp. 1–14 (1993). DOI 10.1007/978-1-4471-3560-9_1. URL http://dx.doi.org/10.1007/978-1-4471-3560-9_1

[67] Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in c programs for fast counterexample detection. In: Proceedings of the 25th International Conference on Computer Aided Verification, CAV'13, pp. 381–396. Springer-Verlag, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-39799-8_26. URL http://dx.doi.org/10.1007/978-3-642-39799-8_26

[68] Kroening, D., Lewis, M., Weissenbacher, G.: Proving Safety with Trace Automata and Bounded Model Checking, pp. 325–341. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-19249-9_21. URL http://dx.doi.org/10.1007/978-3-319-19249-9_21

[69] Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer (2002)

[70] Li, S., Tan, G.: Exception analysis in the java native interface. Science of Computer Programming **89, Part C**, 273 – 297 (2014). DOI http://dx.doi.org/10.1016/j.scico.2014.01.018. URL http://www.sciencedirect.com/science/article/pii/S0167642314000446

[71] Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Principles of Programming Languages (POPL), pp. 611–622 (2011)

[72] Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM **14**(3), 151–165 (1971). DOI 10.1145/362566.362568. URL http://doi.acm.org/10.1145/362566.362568

[73] Mariani, L., Marchetto, A., Nguyen, C., Tonella, P., Baars, A.: Revolution: Automatic evolution of mined specifications. In: Software Reliability Engineering (ISSRE), pp. 241–250. IEEE (2012). DOI 10.1109/ISSRE.2012.14

[74] Mariani, L., Pezzè, M.: A technique for verifying component-based software. Electronic Notes in Theoretical Computer Science **116**(0), 17 – 30 (2005). DOI http://dx.doi.org/10.1016/j.entcs.2004.02.089. URL http://www.sciencedirect.com/science/article/pii/S1571066104052740. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS)

[75] McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification (CAV), LNCS, pp. 123–136. Springer (2006)

[76] de Moura, L., Bjørner, N.: Deciding Effectively Propositional Logic Using DPLL and Substitution Sets, pp. 410–425. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-71070-7_35. URL http://dx.doi.org/10.1007/978-3-540-71070-7_35

[77] Nellis, A., Kesseli, P., Conmy, P.R., Kroening, D., Schrammel, P., Tautschnig, M.: Assisted Coverage Closure, pp. 49–64. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-40648-0_5. URL http://dx.doi.org/10.1007/978-3-319-40648-0_5

[78] Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in Yogi. In: International Conference on Software Engineering (ICSE). Association for Computing Machinery, Inc. (2010). URL http://research.microsoft.com/apps/pubs/default.aspx?id=117670

[79] O'Keeffe, M., Cinnéide, M.: Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice **20**(5), 345–364 (2008)

[80] O'Keeffe, M., Cinnéide, M.: Search-based refactoring for software maintenance. Journal of Systems and Software **81**(4), 502 – 516 (2008)

[81] Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Computer Aided Verification (CAV), LNCS, pp. 773–789. Springer (2013)

[82] Raychev, V., Schäfer, M., Sridharan, M., Vechev, M.T.: Refactoring with synthesis. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pp. 339–354 (2013). DOI 10.1145/2509136.2509544. URL http://doi.acm.org/10.1145/2509136.2509544

[83] Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-Guided Quantifier Instantiation for Synthesis in SMT, pp. 198–216. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-21668-3_12. URL http://dx.doi.org/10.1007/978-3-319-21668-3_12

[84] Shafiei, N., Breugel, F.v.: Automatic handling of native methods in Java PathFinder. In: Proceedings of the 21st International SPIN Workshop. ACM (2014)

[85] Shams, Z., Edwards, S.H.: Reflection support: Java reflection made easy. The Open Software Engineering Journal **7**, 38–52 (2013). DOI 10.2174/1874107X20130422001. URL http://dx.doi.org/10.2174/1874107X20130422001

[86] Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Computer Aided Verification (CAV), pp. 88–105 (2014)

[87] Siefers, J., Tan, G., Morrisett, G.: Robusta: Taming the native beast of the jvm. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, pp. 201–211. ACM, New York, NY, USA (2010). DOI 10.1145/1866307.1866331. URL http://doi.acm.org/10.1145/1866307.1866331

[88] Solar-Lezama, A.: Program sketching. STTT **15**(5-6), 475–495 (2013)

[89] Steimann, F.: Constraint-Based Model Refactoring, pp. 440–454. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-24485-8_32. URL http://dx.doi.org/10.1007/978-3-642-24485-8_32

[90] Steimann, F., Kollee, C., von Pilgrim, J.: A Refactoring Constraint Language and Its Application to Eiffel, pp. 255–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-22655-7_13. URL http://dx.doi.org/10.1007/978-3-642-22655-7_13

[91] Steimann, F., von Pilgrim, J.: Constraint-Based Refactoring with Foresight, pp. 535–559. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-31057-7_24. URL http://dx.doi.org/10.1007/978-3-642-31057-7_24

[92] Sun, M., Tan, G.: Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings, chap. JVM-Portable Sandboxing of Java's Native Libraries, pp. 842–858. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-33167-1_48. URL http://dx.doi.org/10.1007/978-3-642-33167-1_48

[93] SV-COMP 2016: http://sv-comp.sosy-lab.org/2016/

[94] Tan, G.: Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings, chap. JNI Light: An Operational Model for the Core JNI, pp. 114–130. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-17164-2_9. URL http://dx.doi.org/10.1007/978-3-642-17164-2_9

[95] Trudel, M., Furia, C.A., Nordio, M., Meyer, B., Oriol, M.: C to O-O translation: Beyond the easy stuff. 2013 20th Working Conference on Reverse Engineering (WCRE) **0**, 19–28 (2012). DOI http://doi.ieeecomputersociety.org/10.1109/WCRE.2012.12

[96] Visser, E.: Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. Tech. Rep. UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University (2004)

[97] Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003). DOI 10.1023/A:1022920129859. URL http://dx.doi.org/10.1023/A:1022920129859

[98] Wang, T.E., Garoche, P., Roux, P., Jobredeaux, R., Feron, E.: Formal analysis of robustness at model and code level. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC, pp. 125–134 (2016)

[99] Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: Automated Software Engineering (ASE), pp. 231–240 (2006)

[100] Wintersteiger, C.M., Hamadi, Y., Moura, L.: Efficiently solving quantified bit-vector formulas. Form. Methods Syst. Des. **42**(1), 3–23 (2013). DOI 10.1007/s10703-012-0156-2. URL http://dx.doi.org/10.1007/s10703-012-0156-2