Symbolic Execution Equivalence of Heap Modifying Programs

by

Ahmed Abdalla Elbashir Mohammed Elkhair

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Equivalence check of two programs (or two versions of a program) is the problem of deciding if the two programs produce the same values as output for the same input. Symbolic execution is a program analysis technique that executes the programs under analysis using symbolic values. Several approaches and tools utilize symbolic execution to check equivalence. Existing symbolic execution implementations face multiple difficulties including scalability and limited support for language constructs. Current equivalence check approaches and implementations focus primarily on the scalability issue, often by abstracting unchanged parts of the programs under check. Despite the existence of several extensions to symbolic execution that deal with complex data types, current equivalence check implementations are incapable of performing equivalence checks for programs that deal with non-primitive data types.

We present an equivalence checker that can check Java programs with non-primitive method parameters, fields, and return types., including parameterized types. The checker tracks input and output variables (e.g., instance and static fields, parameters, and returns) for changes throughout the symbolic execution of the program, and collects the transformations of those variables. The checker relies on a symbolic execution extension known as Lazy Initialization, to handle non-primitive data types, and extends the technique to further deal with parameterized types by utilizing type signatures. By supporting non-primitive types and more input and output variables, the checker extends the space of possible programs that we can check for equivalence.

Acknowledgements

I would like to thank my supervisor, Karim Ali, for the incredible amount of patience, guidance, and support he provided me. His detailed comments and feedback were invaluable to my work. This work would not have been possible without his help.

I would like to thank my colleagues in the Maple lab for the great company and the various informative and stimulating discussions we had.

My deepest gratitude goes to my family for the unconditional love and support they provided me throughout this journey.

I gratefully acknowledge the funding support provided by the Department of Computing Science and the Faculty of Graduate Studies and Research at the University of Alberta.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal software verification tools provide verifiable guarantees on software behaviour. Such tools often require specialized understanding, may impose a limited set of development languages and frameworks, and require the production of a formal specification or proofs. The costs incurred by these requirements can be acceptable within certain software categories that warrant high correctness assurance. However, for other types of software such costs can be prohibitively high [23].

A more feasible alternative to formal verification is program equivalence. The goal is to check if two programs are functionally equivalent. Instead of checking a program against a formal specification, an equivalence checker relies on another program instead. This alleviates the costs associated with writing a formal specification, and the need for specialized languages or frameworks for writing the software.

Generally, the problem of deciding whether two programs are equivalent is undecidable. Deciding equivalence of two programs is to decide non-trivial properties about the programs, which is undecidable according to Rice's Theorem [29]. The equivalence check process involves the exploration of the state spaces of the programs. Such space might be large or even infinite, hence affecting the scalability of the checking process. However, equivalence checkers may exploit the commonalities between the programs to render the problem tractable [17]. Additionally, checking the equivalence of closely related programs or successive versions of the same program (or its components) is of

practical importance, as exemplified by the prevalent use of regression tests in software engineering.

One program analysis technique that is used to explore the state space of a program is symbolic execution. In concrete execution, a program runs on specific values as its inputs. While, in symbolic execution, input variables are initialized to symbols. These symbols do not represent specific values, but rather a set of values that can comprise the entire range of the type of the corresponding variable. Consequently, symbolic execution explores all feasible paths of a program. Uses of symbolic execution include checking if a program violates certain properties and test case generation [5], [21]. Symbolic execution may also be used to generate symbolic summaries that characterize the behaviour of the program. Program equivalence, in combination with a decision procedure, may utilize these summaries to perform equivalence check [27].

As a state space exploration technique, symbolic execution suffers from the state space explosion problem [1]. As a result, the main focus for the design of numerous symbolic execution equivalence checkers has been how to control the state search domain without sacrificing the completeness of the analysis. However, due to lack of support of some language features, those checkers can operate on a limited set of programs. For Java programs checkers, these features include: dynamic data structures, methods that take input or produce output on multiple fields, and parameterized types [4], [27], [33].

To address these limitations, this thesis presents HeapChecker: an equivalence checker for Java programs. As an input, HeapChecker takes two methods and as an output states whether they are equivalent or not. HeapChecker relies on Symbolic PathFinder (SPF) [25], a symbolic executor that supports dynamic data structures through a technique known as Lazy Initialization. HeapChecker builds on SPF by generating summaries that include representations for those structures that may then be checked by a satisfiability solver. During execution, HeapChecker identifies writes to fields and parameters and encodes them in the symbolic summaries. Since SPF executes Java bytecode, it lacks support for parameterized types due to type erasure. To recover those types during its analysis, HeapChecker uses type signatures

from the method's attributes table.

By adding support to this set of features, HeapChecker adds heap modifying programs to the set of programs that can be checked for equivalence via symbolic execution. The hypothesis we make and investigate in this thesis is that HeapChecker represents heap-modifying programs in formulae that are checkable for equivalence using satisfiability solvers with equality and uninterpreted functions theory.

1.1 Thesis Contributions

This thesis details the design and implementation of HeapChecker. Its contributions include:

- A methodology for representing data structures in first-order logical formulae for satisfiability solvers over equality with uninterpreted functions and arithmetic theories.

- An implementation, HeapChecker, that checks the equivalence of Java methods that deals with heap related features: user-defined data structures, writes to fields, and parameterized types.

- An evaluation of HeapChecker to investigate the validity of the hypothesis of the thesis. The evaluation details HeapChecker's capabilities and limitations. We find that HeapChecker performance on basic benchmarks with no heap-related features is comparable to previous equivalence checkers. We show that HeapChecker correctly decides equivalence and non-equivalence for programs with heap related features, in the absence of execution paths that are dependent on aliasing check. We show that while HeapChecker's execution times for most heap-related features are within similar ranges to checks on programs with basic features, the execution times do not scale as well when dealing with reference type input in recursive methods.

1.2 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 offers an overview of related work regarding symbolic executors and approaches for program equivalence check. Chapter 3 provides background information on the symbolic execution equivalence check process, starting with symbolic execution, generating symbolic summaries, satisfiability, and ending with an overview of the two frameworks we build HeapChecker on: the Java PathFinder (JPF) model checking framework and the Symbolic PathFinder (SPF) symbolic executor. Chapter 4 details the approach and workings of HeapChecker and how it enables the equivalence check of methods that accept and modify dynamic data structures, write and modify fields and parameters, and accept parameterized type input. Chapter 5 provides an overview of the implementation of HeapChecker and how it builds on SPF and JPF. Chapter 6 details the evaluation of HeapChecker, the benchmarks used to perform the evaluation, and demonstrates the capabilities and limitations of HeapChecker. Finally, Chapter 7 concludes the thesis and offers future research directions.

Chapter 2

Related Work

This chapter provides an overview of previous work on symbolic execution and equivalence checking tools that rely on symbolic execution.

2.1 Symbolic Execution

Several equivalence checking and semantic difference analysis tools rely on symbolic execution to explore the state space of the programs under check [3], [4], [27], [33]. Researchers introduced several symbolic executors that vary in language support, path exploration strategy, and memory modelling [5]. Symbolic executors such as angr [30], BAP [7], and BitBlaze [31] analyze binary code after translating (lifting) it to intermediate language (IR). KLEE [8] operates on code compiled to LLVM IR. Other symbolic executors execute Java bytecode, such as Symbolic PathFinder (SPF) [24] and JDart [22], both built atop Java PathFinder (JPF) [34], which is a model checking framework for Java bytecode programs.

A symbolic executor may need to deal with complex constraints or external code that it cannot symbolically execute. Several symbolic executors such as DART [14], SAGE [15], and S2E [9], alleviate this issue by executing the program both concretely and symbolically, by resorting to concrete execution when dealing with external code or complex constraints.

KLEE, BitBlaze, and BAP implement symbolic memory model, in which the memory addresses are modelled as symbolic values. On the other hand, SPF implements a lazy initialization approach [20], where fields of objects

gets initialized symbolic values when they are first accessed. Kiasan [12] implements different variations of the lazy initialization algorithm and formalized operational semantics. Java StarFinder (JSF) [28] implements symbolic execution with separation logic to handle heap programs. To generate the symbolic summaries necessary for equivalence check, HeapChecker relies on SPF to symbolically execute the programs under check.

2.2 Equivalence Checking

While the problem of deciding the equivalence of two programs is undecidable, the problem can be tractable if a degree of similarity exists between the programs under check [17]. Several equivalence checking tools attempt to utilize this aspect by restricting the analysis to the parts that can be readily identified as different through syntactic means. For example, Regression Verification Tool (RVT) [17] checks the equivalence of C programs by traversing the call graph of the programs under check, and skipping the functions that are syntactically equivalent across the two programs. Thus, it only resorts to checking the semantic equivalence when it encounters functions that are syntactically different. RVT relies on CBMC to check for the semantic equivalence. CBMC [10] implements a Bounded Model Checker for C. It can check the validity of assertions by transforming the program into a static single assignment (SSA) form and conjoining the resulting statements to obtain a logical formula that can be checked using a SAT solver.

Besides equivalence checking, techniques have been proposed to identify and characterize the nature of the change between the programs. One example is Differential Symbolic Execution (DSE) [27], which aims to decide the equivalence of programs under check and also, for programs that are not equivalent, characterize the differences between them. DSE uses a syntactic diff tool to identify the syntactically identical portions of the programs under check. It then abstracts those parts and replaces them with uninterpreted functions, thus avoiding the execution of those common parts. DSE relies on SPF to perform symbolic execution and hence operates on Java bytecode.

ModDiff [33] provides equivalence checking and difference characterization. It follows a similar approach to RVT as it also traverses the call graph and restricts the checks on the syntactically different functions. Furthermore, even for the syntactically different functions, it abstracts the common part and allow for the refinement of those abstracted parts to provide more precise analysis. ModDiff operates on programs written in an IR known as the goto-language, and thus can analyze programs that can be translated to this language.

Proteus [3] performs equivalence checking by restricting the symbolic execution to the execution paths impacted by the differences between the two programs under check. To identify the parts affected by the change, Proteus performs a static analysis to extract control and data dependence information. Proteus uses this information to identify the parts of the programs that are impacted by the change and restrict the equivalence check to those parts. Proteus relies on the KLEE symbolic executor and operates on LLVM code.

To balance the over-approximation introduced by the abstraction of parts of the programs under check with the cost of the equivalence check, ARDiff [4] implements an iterative approach that relies on a number of heuristics to guide the process of refining the abstracted portions of the program.

While the focus of those equivalence checkers is to either improve the completeness or balance the completeness with the precision of the analysis, little attention is given to tackling programs that manipulate dynamically allocated objects. The main focus of HeapChecker is to handle such programs, rather than the issue of managing the amount of equivalence check that a tool needs to perform.

Chapter 3

Symbolic Execution Equivalence Check

This chapter provides background information on the process of checking program equivalence using symbolic execution.

3.1 Symbolic Execution

Executing a program concretely runs the program on a specific set of input values, exploring a single execution path on each run. Alternatively, symbolic execution replaces concrete values with symbols. Having symbols instead of concrete values enables symbolic execution to explore multiple paths that may result from executing different sets of concrete inputs.

Example 3.1.1. Consider the code snippets in Figure 3.1. Methods `abs1()`

```
1  int abs1(int x) {
2    if(x < 0) {
3      result = -x;
4    } else {
5      result = x;
6    }
7    return result;
8  }
```

(a) Version 1 of `abs`.

```
9  int abs2(int x) {
10   if(x == 0) {
11     result = 0;
12   } else if(x > 0) {
13     result = x;
14   } else {
15     result = -x;
16   }
17   return result;
18 }
```

(b) Version 2 of `abs`.

Figure 3.1: An example illustrating syntactically different methods with the same functionality.

and `abs2()` have the same functionality, which is the computation of the absolute value of the input. The difference between the two is that `abs1()` adds an additional if-condition (Line 10) that checks whether the input is equal to zero and changes the order of the other input checks (Line 12 and Line 14). The change does not affect the functionality. We refer to each statement by its line number.

When executing a method, the symbolic execution engine records for each execution path a path condition and path transformations. An execution path of a method is a sequence of statements that starts at the entry point of the method, and ends at an exit point. The method `abs1()` in Figure 3.1a has two execution paths. One path comprises Line 3 and Line 7, which we refer to as $p_{(3,7)}$. The second path is composed of Line 5 and Line 7, which we label $p_{(5,7)}$.

For a method execution to follow a certain path, the path condition needs to be satisfiable, that is, there exists a valuation of the variables of the path condition that renders it true. A feasible execution path is a path whose path condition is satisfiable.

Definition 3.1.2. A path condition ($cond(p)$) of an execution path p is a formula that encodes constraints on variables that statements in p read. It is a conjunction of the conditional expressions (or their negations) of branch statements in p. The variables in the path condition must satisfy those conditions for an execution to take that path.

For the execution to take the path $p_{(3,7)}$, the conditional expression $x < 0$ needs to be true. Therefore, the path condition for $p_{(3,7)}$, $cond(p_{(3,7)})$, is $x < 0$, while for $p_{(5,7)}$, which follows the else branch, $cond(p_{(5,7)}) \equiv x \geq 0$.

Path transformations map program variables to the values (symbolic or concrete) obtained throughout the execution of the method.

Definition 3.1.3. Path transformations ($transform(p)$) encode the effect of the execution path p on the variables that m writes to.

$$transform(p) := \bigwedge v \in Write(p) \implies v = f_{p,v}(V_{m,in}) \tag{3.1}$$

Write(p) denotes the set of variables the execution path p writes to. The function $f_{p,v}$ maps the set of input variables $V_{m,in}$, that constitutes the input to the method m, to the value that p computes and writes to v [33].

Each execution path of `abs1` in Figure 3.1a takes a single variable (`x`) as an input and a single value (the `return` value) as an output. Therefore, the sets *Write*(p) (for both execution paths), and $V_{m,in}$ are $\{return\}$ and $\{x\}$, respectively. It follows that, $transform(p_{(3,7)}) \equiv (return = -x)$ and $transform(p_{(5,7)}) \equiv (return = x)$.

One use of symbolic execution is to generate a symbolic summary for the program or method under analysis. A method symbolic summary is a formula that captures the behaviour of the method. A method's feasible execution paths constitute all of the method's execution possibilities. Therefore, the method's symbolic summary is comprised of the symbolic summaries of its feasible paths; the symbolic summary of each execution path combines its path condition and transformations.

Definition 3.1.4. A symbolic summary of a path p is the conjunction of the path condition ($cond(p)$) and transformations ($transform(p)$) resulting from that path.

$$s_p := cond(p) \wedge transform(p) \tag{3.2}$$

The conjunctions of path conditions and transformations of $p_{(3,7)}$ and $p_{(5,7)}$ yield their symbolic summaries:

$$\begin{aligned} s_{p_{(3,7)}} &\equiv cond(p_{(3,7)}) \wedge transform(p_{(3,7)}) \\ &\equiv x < 0 \wedge return = -x \\ s_{p_{(5,7)}} &\equiv cond(p_{(5,7)}) \wedge transform(p_{(5,7)}) \\ &\equiv x \geq 0 \wedge return = x \end{aligned}$$

Definition 3.1.5. The symbolic summary of a method m is the disjunction of the symbolic summaries of the feasible execution paths of m.

$$S_m := \bigvee p \in Feasible(p) \implies s_p \tag{3.3}$$

Where *Feasible*(m) is the set of feasible execution paths in m.

The disjunction of the summaries of the execution paths $abs1()$ (i.e., $p_{(3,7)}$ and $p_{(5,7)}$) compose its symbolic summary:

$$\begin{aligned} S_{abs1} &\equiv s_{p_{(3,7)}} \vee s_{p_{(5,7)}} \\ &\equiv (x < 0 \wedge return = -x) \vee (x \geq 0 \wedge return = x) \end{aligned} \tag{3.4}$$

The symbolic summary of `abs2()` is composed of the summaries of its three execution paths:

$$S_{abs2} \equiv (x = 0 \wedge return = 0) \vee (x > 0 \wedge return = x) \vee (x < 0 \wedge return = -x) \tag{3.5}$$

3.2 Lazy Initialization

The symbolic execution of a method that takes an input of primitive type assigns a single symbolic value to the input prior to the execution. However, if the method accepts a reference type (non-primitive) input, then symbolic execution requires multiple symbolic values to represent the reference to the object in addition to each one of its fields. One way to handle methods that access reference fields and parameters, is to lazily initialize those variables [20]. The algorithm works by initializing a reference field or parameter and its fields when the method first accesses the field or parameter. The algorithm explores initialization possibilities for each variable to:

- `null`
- an object of the same type that has been previously initialized
- a new object

Example 3.2.1. The class `LinkedList` in Figure 3.2 is an implementation of a linked list of integers. It has two instance fields `next` and `content`, and a single method `getNextInt()` that retrieves the integer stored in the next node

```
19 class LinkedList {
20   LinkedList next;
21   int content;
22
23   int getNextInt() {
24     if(this.next == null) {
25       return 0;
26     } else {
27       return next.content;
28     }
29   }
30 }
```

Figure 3.2: An example demonstrating lazy initialization.

if it exists, or zero otherwise. When the symbolic executor encounters the first read of the field `this.next` (Line 24), it initializes `this.next` to one of the three initialization options: `null`, which results in returning zero; another object of the same type, which in this case is the current object; or a new object and returning the integer field of that object. After the symbolic executor finishes executing the path resulting from a specific initialization option, it backtracks and explores another one until it exhausts all feasible paths.

The summaries that correspond to each path are:

Initializing `this.next` to null:

$$s_{null} \equiv (this.next = null \wedge return = 0) \tag{3.6}$$

Initializing `this.next` to an object of the same type (the current object in this case):

$$s_{next=this} \equiv (this.next = this \wedge this.next \neq null \wedge return = this.content) \tag{3.7}$$

Initializing `this.next` to a new object:

$$s_{next\neq this} \equiv (this.next \neq this \wedge this.next \neq null \wedge return = this.next.content) \tag{3.8}$$

3.3 Equivalence check

We base the definition of equivalence on the definition of full equivalence introduced by Godlin and Strichman [16].

Definition 3.3.1. Two methods m_1 and m_2 are equivalent if and only if for all possible values of input variables, they terminate and they produce the same values on the same output variables.

A symbolic method's summary fully characterizes the input-output behaviour of the method. Therefore, an equivalence checker may utilize the symbolic summaries of methods to decide on their equivalence. To check whether two methods m_1 and m_2 are equivalent or not, we check the validity of the equivalence verification condition:

$$C_{equiv}(m_1, m_2) := S_{m_1} \iff S_{m_2} \tag{3.9}$$

Where S_{m_1} and S_{m_2} are the symbolic summaries of m_1 and m_2. The two methods are equivalent if and only if the equivalence verification condition is valid. For the two versions of `abs` in Figure 3.1, the equivalence verification condition is composed of the summaries in Equations (3.4) and (3.5):

$$\begin{aligned} C_{equiv}(abs1, abs2) &:= S_{abs1} \iff S_{abs2} \\ &= (x < 0 \wedge return = -x) \vee (x \geq 0 \wedge return = x) \iff \\ &(x = 0 \wedge return = 0) \vee (x > 0 \wedge return = x) \vee (x < 0 \wedge return = -x) \end{aligned} \tag{3.10}$$

3.4 Satisfiability

The problem of propositional satisfiability (SAT) [6] is to decide whether a propositional formula is satisfiable or not. A formula is satisfiable if and only if there exists a combination of true or false values that can be assigned to the boolean variables of the formula such that the formula evaluates to true. If no such combination exists, then the formula is unsatisfiable. Another related concept is validity. A formula is valid if and only if for every combination of true or false value the formula is true. Validity and satisfiability are related. A formula is valid if and only if its negation is unsatisfiable.

A Satisfiability Modulo Theories (SMT) [6] solver extends SAT solving so that it can check the satisfiability of more expressive logic, such as first-order

logic. SMT solvers achieve this by incorporating specific theory solvers, in additional to the propositional SAT solver. One such theory is the theory of equality with uninterpreted functions (EUF) and integer arithmetic. A SAT modulo EUF can, for instance, check formulae such as Equations (3.6), (3.7), and (3.8). With both EUF and integer arithmetic, an SMT solver can check the satisfiability (and validity) of the verification condition (3.10). Since Equation (3.10) is valid, it follows that `abs1()` and `abs2()` are equivalent.

Symbolic executors rely on SMT solvers to determine the feasibility of a path by checking the satisfiability of its path condition. Equivalence checker may use SMT solvers to check the validity of the equivalence verification condition. Several off-the-shelf SMT solvers exist such as Microsoft's Z3 theorem prover [11], which supports several theories such as EUF, arithmetic, and bit-vectors.

3.5 Java PathFinder and Symbolic PathFinder

Java PathFinder (JPF) is an extensible model checking framework for Java programs [19], [34]. JPF contains a modular implementation of the Virtual Machine for Java bytecode. This implementation enables the user to re-define the semantics of the Java bytecode instructions, and provide their own interpretation of Java bytecode instructions. JPF provides a search component that enables the JPF VM to systematically explore the state space of the program under check. One key feature of JPF is the Attribute System. In addition to the normal data flow (e.g., operands on the stack and objects in the heap) in JPF VM that is similar to the normal VM, JPF provides a storage system that allows additional attributes to be attached to the data. The Attribute System can be used to propagate symbolic values during the execution.

Symbolic PathFinder (SPF) is a symbolic executor for Java bytecode programs [25], [32]. It extends JPF by overriding some of the bytecode instructions in the JPF VM and providing a symbolic interpretation for those instructions. SPF utilizes the Attribute System of JPF to enable the propagation of symbolic data. SPF supports primitive data types as well as data structures

through lazy initialization. Applications of SPF include test case generation and error detection.

Chapter 4

Overview of HeapChecker

This chapter details the approach that HeapChecker follows to enable symbolic execution equivalence check of methods that operate on non-primitive data types. Figure 4.1 provides an overview of HeapChecker's equivalence check process.

4.1 Fields Transformations

Sound equivalence check requires symbolic summaries that fully characterize the behaviour of the methods under check. Thus, we need to include transformations of all the fields that the methods have written to. HeapChecker builds and adds the transformations for fields during and after symbolic execution.

Prior to the beginning of executing the instructions of a method m under check, HeapChecker initializes all of m's parameters, and the containing object's fields to symbols. If during the symbolic execution of a path p in m HeapChecker encounters a write to a field f, it checks whether the field is symbolic or not. A symbolic field constitutes an input to the method under check, and a write to that field means that the field is also part of m's output. If f is symbolic, HeapChecker adds f to a set F_p that consists of fields that p has written to, if it has not been added before. If HeapChecker encounters a return statement, it creates and adds a symbol *return* to F_p to represent the value the method returns, if it exists.

Example 4.1.1. Figure 4.2 depicts two equivalent versions of the method `m()`.

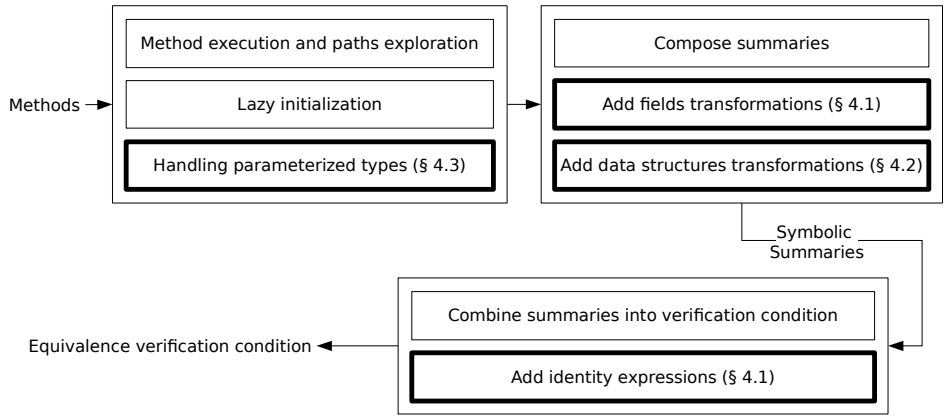


Figure 4.1: Workflow of HeapChecker up to the generation of verification condition. Thick borders indicate the new components in HeapChecker.

```
31 void m1() {
32   if (x == 0) {
33     return;
34   } else {
35     y = y + x;
36   }
37 }
```

(a) Version 1 of `m()`.

```
38 void m2() {
39   y = y + x;
40 }
```

(b) Version 2 of `m()`.

Figure 4.2: An example illustrating the scenario when two versions of method `m()` are equivalent while their summaries are inequivalent if they only contain changed fields transformations.

Version 1 has two execution paths (indicated by lines numbers): p_{33} and p_{35}, while version 2 has a single execution path p_{39}. Since both p_{35} and p_{39} write to the same field y, each one of $F_{p_{35}}$ and $F_{p_{39}}$ have the single element y. The path p_{33} writes to no field, so $F_{p_{33}}$ is the empty set.

At the end of the execution path p, indicated by a return instruction, HeapChecker builds the path transformations and summaries. For each field f in F, HeapChecker retrieves the value of the field f, v_f, from the heap, builds an expression $f_{out} = v_f$, and adds the expression as a conjunct to the transformations formula of p:

$$\mathit{transform}(p) := \mathit{transform}(p) \wedge (f_{out} = v_f) \tag{4.1}$$

The symbol f_{out} represents the value of f at the end of the execution of p, while f_{in} represents the value of f at the beginning. Thus, $\mathit{transform}(p_{33})$ is an empty conjunction, while both $\mathit{transform}(p_{35})$ and $\mathit{transform}(p_{39})$ equals $y_{out} = y_{in} + x_{in}$.

Consequently, we get the summaries for `m1` and `m2` as follows:

$$S_{m_1} \equiv (x_{in} = 0) \vee (x_{in} \neq 0 \wedge y_{out} = y_{in} + x_{in}) \tag{4.2}$$

$$S_{m_2} \equiv (y_{out} = y_{in} + x_{in}) \tag{4.3}$$

After the termination of the symbolic execution of both methods under check and the generation of their summaries, HeapChecker proceeds to build the equivalence verification condition. While the summaries' transformations include all changed fields, they do not correctly characterize execution paths where those same fields remain unchanged. Despite the fact that `m1` and `m2` are equivalent, the verification condition $S_{m_1} \iff S_{m_2}$ is not valid, since S_{m_1} does not assert that y remains unchanged when x is zero. Rather, it implies that when x is zero, y can take any value, as opposed to retaining its initial value.

To resolve this, HeapChecker adds an identity expression $f_{out} = f_{in}$ to the transformations of each execution path p where f is unchanged, for each field f such that $f \in F_q$ for some other execution path q that writes to f.

$$transform(p) := (\bigwedge f \notin F_p \wedge f \in F_q \wedge p \neq q \implies (f_{out} = f_{in})) \wedge transform(p) \tag{4.4}$$

For the path p_{33} in m, $y_{out} = y_{in}$ is added to $transform(p_{33})$, yielding the following summary for m_1:

$$S_{m_1} \equiv (x_{in} = 0 \wedge \boldsymbol{y_{out} = y_{in}}) \vee (x_{in} \neq 0 \wedge y_{out} = y_{in} + x_{in}) \tag{4.5}$$

The resulting summaries fully characterize the methods under check. HeapChecker composes the verification condition and uses an SMT solver to check its validity.

4.2 Dynamic Data Structures

If the right-hand side of a transformation expression is a dynamic data structure, then HeapChecker encodes the object in the summary by recursively adding to the transformation formula, a transformation expression for each of its fields.

For each field f that is assigned a reference to a dynamic data structure in an execution path p, HeapChecker adds a conjunction of transformations $transform(p, f)$ that represents the assignment of the reference of the structure to f, and recursively, the values assigned to the fields of the structure.

For each field f that p writes to, HeapChecker generates $transform(p, f)$ using the following recursive formula:

$$transform(p, f) \equiv \begin{cases} f_{out} = v_f & f \text{ is primitive} \\ f_{out} = null & f \text{ is null} \\ f_{out} = f'_{out} \wedge (\bigwedge f_i \in Fields(f) \implies transform(p, f'_{out}.f_i)) & f \text{ is not null} \end{cases} \tag{4.6}$$

Where $Fields(f)$ is the set of fields of the data structure assigned to f.

The formula is a generalization of fields' transformations to both primitive and non-primitive fields. If the field f is of a primitive type, then the formula reduces to $f_{out} = v_f$, which is the rightmost conjunct in Equation (4.1). For a

```
42 class Tree {
43   Tree left, right;
44   int content;
45
46   public Tree(Tree l, Tree r,
         int c) {
47     left = l;
48     right = r;
49     content = c;
50   }
51 }
```

```
54 class TreesBuilder {
55   Tree t1, t2;
56   public void buildTrees(int x) {
57     Tree small_t = new Tree(null,
           null, x);
58     Tree medium_t = new Tree(small_t
           , null, x);
59     if (x > 0) {
60       t1 = small_t;
61       t2 = medium_t;
62     } else {
63       t1 = medium_t;
64       t2 = medium_t;
65     }
66   }
67 }
```

Figure 4.3: An example illustrating how HeapChecker encodes data structures transformations.

non-primitive null assignment, HeapChecker adds an expression $f_{out} = null$. For the non-primitive non-null case, where f is assigned a data structure object, HeapChecker introduces a new symbol, f'_{out}, to designate that object. It adds an expression $f_{out} = f'_{out}$ to represent the assignment. Then HeapChecker recursively adds the transformations of each field f_i of the object assigned to f. HeapChecker assigns each field the symbol $f'_{out}.f_i$, where the dot '.' represents the field access relation between f'_{out} and f_i. The naming pattern that HeapChecker follows assigns f'_{out} the same name as f_{out}, but appends the suffix `"_local"` to it.

HeapChecker builds the transformations for each field following this pattern. Thus if multiple fields are assigned the same object, HeapChecker will add separate transformations for each field. This approach ensures that the summaries remain consistent regardless of the order of the assignment in the methods under check. To indicate that multiple fields (e.g., f, g, and h) are assigned the same object, HeapChecker adds additional expressions to indicate so:

$$f'_{out} = g'_{out} \wedge f'_{out} = h'_{out} \tag{4.7}$$

Example 4.2.1. The method `buildTrees()` in Figure 4.3 takes an integer `x` as an argument, builds two trees (Lines 57–58), and, based on the value of `x`, assigns the built trees to instance fields, `t1` and `t2` (Lines 60–61 and Lines 63–64).

The following is the formula HeapChecker produces for $transform(p_{consequent}, t_1)$; the transformations of the field `t1` for the execution path that follows the consequent branch (`x>0`) of the `if` statement (Lines 59–61). In other words, the effects of the statements `Tree small_t = new Tree(null, null, x);` and `t1 = small_t;`:

```
_this.t1_out = _this_t1_out_local &&

_this_t1_out_local.content = x &&
_this_t1_out_local.right is NULL &&
_this_t1_out_local.left is NULL
```

The term `_this` denotes a reference to the current object. Consequently, the terms `_this.t1_out` and `_this.t2_out` reference the fields `t1` and `t2` of the current object. The expression in Line 68 denotes the assignment of the reference of `small_t` to the field `t1`. Thus, `_this_t1_out_local` represents `small_t`. Lines 70–72 denote the transformations of the fields of `_this_t1_out_local` (i.e., `small_t`).

While for $transform(p_{consequent}, t_2)$, HeapChecker produces the following:

```
_this.t2_out = _this_t2_out_local &&

_this_t2_out_local.content = x &&
_this_t2_out_local.right is NULL &&
_this_t2_out_local.left = _this_t2_out_local_left_local &&

_this_t2_out_local_left_local.content = x &&
_this_t2_out_local_left_local.right is NULL &&
_this_t2_out_local_left_local.left is NULL
```

The expression at Line 73 denotes the assignment of the reference of `medium_t` to the field `t2`. Therefore, `_this_t2_out_local` represents `medium_t`.

Lines 75–77 denote the transformations of the fields of `_this_t2_out_local` (i.e., `medium_t`). Lines 79–81 encode the transformations of `_this_t2_out_local_left_local` (i.e., `this.t2.left`). The object `_this_t2_out_local_left_local` is the same object as `_this_t1_out_local`, which is `small_t`. To represent this alias-

ing fact, HeapChecker adds the following expression to indicate that both `this.t1` and `this.t2.left` are assigned references to the same object:

```
_this_t1_out_local = _this_t2_out_local_left_local
```

The conjunction of these three formulae comprises the formula $transform(p_{consequent})$, that is, the transformations of fields that the consequent branch of the if-statement (Lines 59–61).

Algorithm 1 Encodes dynamic data structure transformations.

Input:

1: f: name of the field that is assigned the object.
2: r: reference to the object on the heap.
3: *names*: a function that maps objects' references to names. Returns null if a reference is not assigned a name.

Output:

4: *transforms*: fields transformations of the object.
5: **procedure** EncodeTransforms($f, r, names, transforms$)
6: **if** r is null **then**
7: $transforms \leftarrow transforms \wedge (f = \texttt{null})$
8: **return**
9: **end if**
10: $f' \leftarrow f +$ "*_local*"
11: $transforms \leftarrow transforms \wedge (f = f')$
▷ Check if there are previous names for the same object
12: $a \leftarrow names(r)$
13: **if** a is null **then**
14: $names(r) \leftarrow f'$
15: **else**
16: $transforms \leftarrow transforms \wedge (a = f')$
17: **end if**
18: **for** $f_i \in Fields(f)$ **do**
19: $v_f \leftarrow getFieldValue(f_i)$
20: **if** v_i is a concrete value and primitive or v_i is symbolic **then**
21: $transforms \leftarrow transforms \wedge (f_i = v_f)$
22: **else** ▷ Field is non-primitive
23: EncodeTransforms(f_i, v_i, *names*, *transforms*)
24: **end if**
25: **end for**
26: **end procedure**

To generate the recursive formula in Equation (4.6), HeapChecker implements Algorithm 1. EncodeTransforms takes four input parameters.

The first one, f, is the name of a field that has as a value a reference to an object on the heap. The second parameter, r, is the object reference that the field f is assigned its reference. The third parameter, *names*, is a function that maps each encountered object reference to its variable name. The algorithm uses *names* to record the first name associated with a reference so that it can add aliasing expressions if it encounters more instances of the same reference. The function *names* returns null for any reference that is not assigned a name. The last parameter, *transforms*, is where the algorithm writes its output. When EncodeTransforms terminates, *transforms* contains all the transformations of the fields of r.

After HeapChecker finishes the symbolic execution of a path p, for each field f_i in F_p that is assigned a reference r_i for dynamic data structure, HeapChecker invokes EncodeTransforms$(f_i, r_i, names, transforms)$. Prior to the first call to EncodeTransforms, HeapChecker initializes *transforms* to the empty formula and *names* to return null on all references.

EncodeTransforms first checks if the reference r assigned to the field f is `null` (Line 6). If that is the case, EncodeTransforms adds a `null`-equality expression f = null as a conjunct to the transformations formula *transforms* and then terminates. Otherwise, in Line 10, EncodeTransforms creates a symbol f' that represents the assigned reference by appending the suffix `_local` to the field's name. It then, in Line 11, adds an equality expression $f = f'$ that denotes the assignment of the reference to the field. Afterwards, EncodeTransforms checks if a previous call to EncodeTransforms has added transformations for the same object's reference and created a name for it. It does this check by calling the function *names* on the reference r in Line 12. The function *names* is a mapping from references to the names EncodeTransforms assigned to them. If a name a exists for the reference r (Line 15), EncodeTransforms adds an aliasing expression $a = f'$ to indicate so. If no name exists (Line 13), EncodeTransforms redefines *names* to include the newly created name f'. After that, EncodeTransforms iterates on each field f_i of the object assigned to f and adds the transformations associated with each field f_i (Lines 18–25). The value v_i

that the field f_i holds can be one of three possibilities: a symbolic value, a primitive concrete value, or a non-primitive concrete value. For the first two possibilities, EncodeTransforms adds an equality expression denoting the assignment of the field to the value (Line 21). While for the third possibility, where v_i is a reference to a data structure, EncodeTransforms recursively calls itself to add all fields of the object v_i references (Line 23).

4.3 Parameterized Types

HeapChecker relies on SPF to symbolically execute method with symbolic non-primitive data types through lazy initialization. Upon accessing, on an execution path p, a symbolic field or parameter f of a non-primitive type T for the first time, SPF branches into $3+n$ execution paths, where n is the number of fields of type T SPF has previously initialized to symbolic references to distinct objects. Each execution path follows a different initialization choice with respect to f:

1. assigning f null.
2. assigning f a previously distinctly initialized symbol g of type T.
3. assigning f a new distinct symbol f' followed by assigning symbols to each of f''s fields.

For lazy initialization to work correctly, it needs to acquire the correct type T of the field f. If T is not available or incorrect, then lazy initialization cannot correctly build the set of fields of type T necessary for initialization option number 2. Additionally, it cannot retrieve the set of fields of f' if it cannot correctly establish its type.

Since SPF executes Java bytecode, arguments to parameterized types are erased, and have the type `java.lang.Object`, if they are unbounded and their upper bound if they are bounded. When it comes to the second initialization option type erasure can cause imprecise summaries generation in the following cases:

- During the lazy initialization of a field that has a variable type that had been erased into the type `java.lang.Object` during compilation. Consequently, multiple fields can have the type `java.lang.Object` during the execution, due to type erasure. As a result, SPF forks spurious execution paths to accommodate the aliasing possibilities between these fields. These execution paths are spurious because the compile-time type safety imposed by the parameterized types guarantees that such aliasing is not possible.

- During the lazy initialization of a parameterized type field or parameter that had its type arguments erased into the type `java.lang.Object`. Multiple fields or parameters can have the same generic type, yet different type arguments. However, type erasure renders these fields or parameters to have the same type, because the type arguments had been erased. Similar to how SPF handles erased variable types, SPF will fork spurious execution paths to accommodate the aliasing possibilities between these fields or parameters.

For the third initialization option, a field f of a variable type with a type argument T that is erased into the type `java.lang.Object` or its upper bound means that SPF will not be able to symbolically initialize the fields of f that are specific to the type T. This could result in SPF treating those fields as concrete values and missing execution paths and fields' transformations.

To remove potential imprecise paths resulting from erroneous aliasing possibilities and avoid the omission of execution paths or fields' transformations, HeapChecker constrains lazy initialization of fields and parameters that are of a parameterized type and variable types to the type invocation and the arguments to the type invocation. While the compiler erases type arguments for types invocations in the code, it retains those type arguments in the method signature, which HeapChecker has access to through JPF.

HeapChecker parses the signature of the method under check and extracts the type arguments for parameterized types. HeapChecker then looks up the generic type definition of the parameterized type, maps each type ar-

```
83 class TestPair {
84   String strField;
85   int intField;
86   boolean changed;
87
88   void setField(Pair<String,
         Integer> t, int choice) {
89     if (choice == 0) {
90       strField = t.elem1;
91     } else if (choice == 1) {
92       intField = t.elem2;
93     }
94
95     changed = (choice == 0) || (
           choice == 1);
96   }
97 }
```

(a) Accessing fields of a generic-type parameter.

```
98 aload_0
99 aload_1
100 getfield  #2 // Field Pair.elem1
        :Ljava/lang/Object;
101 checkcast #3 // class java/lang/
        String
102 putfield  #4 // Field strField:
        Ljava/lang/String;
```

(b) JVM bytecode for Line 90, `String` is erased.

```
103 aload_0
104 aload_1
105 getfield #5 // Field Pair.elem2:
        Ljava/lang/Object;
106 checkcast #6 // class java/lang/
        Integer
107 invokevirtual #7 // Method java/
        lang/Integer.intValue:()I
108 putfield #8 // Field intField:I
```

(c) JVM bytecode for Line 92, `Integer` is erased.

Figure 4.4: An example illustrating the erasure of type arguments in Java programs.

gument to its parameter, and then resolves the type of each field or parameter that has a parameterized or a variable type.

Example 4.3.1. As an example for the first case in the second initialization option (fields with variable types), consider the method `setField()` in Figure 4.4a. The method has two parameters. The first parameter is an object `t` that has the parameterized type `Pair<String, Integer>`. This type is an invocation of the generic type `Pair<T, S>`, which has two fields: `elem1` of variable type T and `elem2` of variable type S. During compilation, the Java compiler erases the type arguments of parameterized types. In this example, after compilation, the fields of object `t` (`t.elem1` and `t.elem2`) assumes the type `java.lang.Object`, as Figure 4.4b and Figure 4.4c show. Consequently, SPF will falsely fork an execution path in which `t.elem1` and `t.elem2` are aliases. However, HeapChecker extracts each type argument (`String` and `Integer`) and assigns it to the field that corresponds to the type parameter

taking the argument. Therefore, HeapChecker assigns `java.lang.String` to `t.elem1`, and `java.lang.Integer` to `t.elem2`. This approach prevents the symbolic execution engine from forking an imprecise path where `t.elem1` and `t.elem2` are aliases.

Algorithm 2 Finds possible aliases for an object of a parameterized type.

Input:

1: t: a parameterized type $Type < arg_1, \ldots, arg_n >$, with type definition $Type < T_1, \ldots, T_n >$.
2: H: the set of objects allocated on the heap.

Output: S: a set of objects with parameterized types that are subtypes of t.

3: **procedure** GetPossibleAliases(t, H)
4: $t_{args} \leftarrow getTypeInvocationArgs(t)$ ▷ $< arg_1, \ldots, arg_n >$
5: $t_{params} \leftarrow getTypeDefinitionParams(t)$ ▷ $< T_1, \ldots, T_n >$
6: $S \leftarrow \emptyset$
7: **for** $object \in H$ **do**
8: $t' \leftarrow getType(object)$ ▷ $Type' < arg'_1, .., arg'_m >$
9: $t'_{args} \leftarrow getTypeInvocationArgs(t')$ ▷ $< arg'_1, \ldots, arg'_m >$
10: $t'_{params} \leftarrow getTypeDefinitionParams(t')$ ▷ $< T'_1, \ldots, T'_m >$
11: **if** t' extends t **then**
12: $t'_{result} \leftarrow t_{params}$
13: **for** $i \leftarrow 1, m$ **do**
14: $t'_{result} \leftarrow t'_{result}.replace(t'_{params}[i], t'_{args}[i])$
15: **end for**
16: **if** $t'_{result} = t_{args}$ **then**
17: $S \leftarrow S \cup \{object\}$
18: **end if**
19: **end if**
20: **end for**
21: **return** S
22: **end procedure**

Algorithm 2 details the algorithm for finding possible aliases for an object with a parameterized type. As an input, the algorithm takes a parameterized type t that has the form $Type < arg_1, \ldots, arg_n >$, and the set H of objects allocated on the heap. The type t is an instantiation of the generic type definition $Type < T_1, \ldots, T_n >$. The algorithm's output is the set of all objects with types that are subtypes of t. First, the algorithm parses the type t to extract the type arguments $(arg_1, \ldots, arg_n)$ (Line 4), and the type definition to extract the type parameters $(T_1, \ldots, t_n)$ (Line 5). Then, the algorithm iterates

over all objects with parameterized types in the heap for the current execution path. For each object, the algorithm parses its type and type definition for arguments $(arg'_1, \ldots, arg'_m)$ (Line 9) and parameters $(T'_1, \ldots, T'_m)$ (Line 10). Then it checks if the object's type extends t (Line 11). If yes, the algorithm then obtains the object's parameterized supertype. The algorithm obtains the supertype (t'_{result}) by replacing in the type parameter sequence $T_1, \ldots, T_n$ each type parameter T'_i with its corresponding argument arg'_i (Line 14). If the resulting supertype is t, the algorithm adds the object to the possible aliases (Line 17). This is because in Java, a parameterized type t is a subtype of a parameterized type t' if and only if the generic type definition of t extends the generic type definition of t', and both types have the same type argument for each common type parameter [18]. The algorithm terminates when it finishes iteration over all objects on the heap and returns the set of possible aliases.

Chapter 5

Implementation

This chapter provides an overview of the implementation of HeapChecker and how it extends and integrates with Java PathFinder (jpf-core) [19], [34] and Symbolic PathFinder (jpf-symbc) [25], [32]. HeapChecker consists of three major subsystems: SymbolicVerifier, a modified version of jpf-symbc, and jpf-core. Figure 5.1 shows these subsystems and their major constituent classes. Each of the following sections detail the major classes of each subsystem. Since both JPF and SPF contain a substantial number of classes, we only include classes that are relevant to the production of symbolic summaries, and to the novel functionality that HeapChecker provides.

5.1 SymbolicVerifier

SymbolicVerifier contains the entry point of the system and is responsible for receiving the input, preprocessing and compiling the program files containing the methods to be checked, configuring and starting SPF and JPF, collecting the summaries, composing the equivalence verification condition, and checking the validity of the verification condition.

The main classes under SymbolicVerifier are:

- `MethodsEquivalenceChecker`. This is the class that receives the input of the system. The input consists of the paths to the Java program files containing the methods to be checked, and the name of the method. `MethodsEquivalenceChecker` instantiates two instances of `MethodSummarizer`, one for each method. `MethodsEquivalenceChecker` calls the method

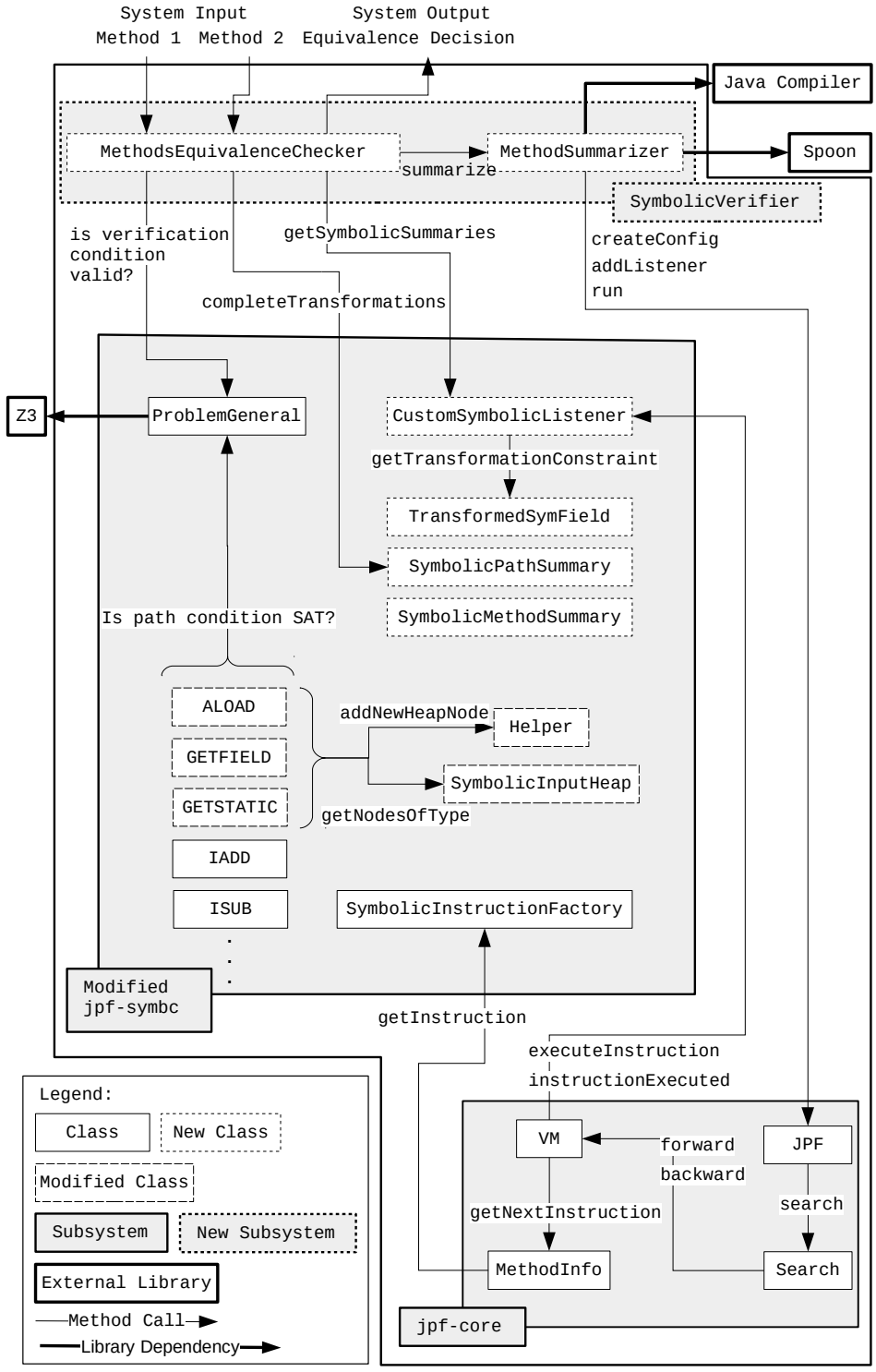


Figure 5.1: The diagram illustrates the organization of the subsystems of HeapChecker and the major classes within each component.

`summarize` on each instance to obtain the summaries of the two methods. Afterwards, `MethodsEquivalenceChecker` extracts all the transformed fields from both summaries for all symbolic paths' summaries. Then, for each symbolic path summary, it invokes the method `completeTransformations`, to add to summary the identity expressions, as described in Equation (4.4). After that, `MethodsEquivalenceChecker` constructs the equivalence verification condition in Equation (3.9). Finally, it checks the validity of verification condition, using the SMT interface provided by SPF.

- `MethodSummarizer`. This class receives a path to a Java program file and a name of a method in the file to be checked. If multiple methods with the same name exist, it chooses the first method declared in the file. `MethodSummarizer` pre-processes the file by adding a wrapper method that instantiates an object and calls the target method, if it is an instance methods. This step is necessary because SPF requires having a main or entry method that invokes the target method. `MethodSummarizer` relies on the Spoon library [26] to perform these transformations on the Java program file. After pre-processing, `MethodSummarizer` invokes the Java compiler to compile the Java file to bytecode class file. Following this, `MethodSummarizer` configures and starts JPF. JPF configurations include setting the target class and method, specifying search strategy and search depth, and decision procedure. Finally, `MethodSummarizer` registers `CustomSymbolicListener` as a listener and starts JPF. After JPF terminates, `MethodSummarizer` retrieves the summaries from the listener, returns the generated symbolic summary and terminates.

5.2 Java PathFinder (jpf-core)

The JPF framework handles the search process and the execution of the method under check. JPF provides extensible interfaces and classes, allowing third-party software to override many of JPF's default search and execution behaviour. These interfaces and classes include JVM instructions, search strategies, and property listeners. JPF consists of numerous classes. However,

in this section, we focus on a small subset. Specifically, some of the classes that interact with SPF and SymbolicVerifier:

- `JPF`. This class is JPF's main class and is the entry point through which SymbolicVerifier starts JPF. `JPF` receives configurations detailing the system under test, search strategy, listeners, virtual machine type, instruction factory, and many other configurations. This class initializes JPF virtual machine and starts the search mechanism. It also loads the main class file of the system under test and prepares the class by adding its instructions in accordance to the configured instruction factory.

- `Search`. This abstract class provides methods' declarations that governs how JPF explores the search space of the system under check. One concrete class that extends Search is `DFSearch`, which implements a depth-first search strategy. The basic methods that `Search` provides are `search`, `forward`, and `backtrack`. Both `forward` and `backtrack` instructs the VM to move to the next state or backtrack to a previous one, while the `search` method controls the overall search strategy.

- `VM`. This abstract class represents the virtual machine. The class provides forward and backtrack methods that move the execution one step forward or backwards. The `VM` calls methods of registered listeners during various stages of the execution of the program under test. One such listener is `PropertyListenerAdapter`, which provides methods that JPF VM calls each before or after a particular event during the execution of the program under test. Such events include instruction execution, property violation, class loading, and thread termination.

- `MethodInfo`. This class, and other similar ones such as `ClassInfo`, contains information about the method under execution. Such info include the sequence of instructions in the method and its arguments. JPF VM builds the sequence of instructions in `MethodInfo` according to the configured instruction factory.

5.3 Symbolic PathFinder (jpf-symbc)

SPF extends JPF to provide symbolic execution capability. SPF provides symbolic implementations for JVM instructions and property listeners for the collection of symbolic summaries and exceptions detection. We modify existing SPF classes to support handling parameterized types during lazy initialization and we add new classes to support inclusion of writes to fields in the summaries and representation of dynamic data structures.

The sets of classes that SPF provides include:

- `SymbolicInstructionFactory`. Provides a lookup table for JPF to access SPF's version of the JVM instructions.

- Symbolic implementations of instructions: SPF provides a set of classes that implement a number of the JVM instructions. These include branching instructions, arithmetic instructions, and method invocation instructions. Of particular importance to HeapChecker are the instructions `ALOAD`, `GETFIELD`, and `GETSTATIC`. `ALOAD` loads a reference from a local variable on the stack frame to the operand stack. Local variables in the JVM stack frame corresponds to the current method's parameters. Therefore, `ALOAD` loads a method's parameters on the operand stack. `GETFIELD` loads an instance field from an object onto the operand stack, while `GETSTATIC` loads a static field.

  If one of the three instructions loads a field or parameter that is both symbolic and of a non-primitive type, then it initializes the field to one of the three options of Lazy Initialization. These instructions call `SymbolicInputHeap.getNodesOfType()` to retrieve previously initialized symbols of the same type as the field or parameter currently being initialized. We have modified `SymbolicInputHeap.getNodesOfType()` to account for parameterized types by implementing GetPossibleAliases detailed in Algorithm 2. Whenever SPF creates a new distinct symbol for the field, each of the three instructions invokes `Helper.addNewHeapNode()`, which allocates an object on the heap, and initializes each field of the

symbolic field or parameter to symbolic values. Similar to `SymbolicInputHeap.getNodesOfType` we modify `Helper.addNewHeapNode()` to account for fields of parameterized type, allowing for complete initialization of the fields of the symbolic field or parameter.

- `ProblemGeneral`. This is an abstract class that provides an interface to satisfiability solvers, while abstracting the API differences between the solvers. SPF provides a number of concrete subclasses of `ProblemGeneral` that are specific for particular solvers such as Z3 [11]. SPF uses methods this class provides to check for the satisfiability of paths conditions, while SymbolicVerifier uses it to check for the validity of the equivalence verification condition.

Additionally, we have added the following classes to SPF:

- `CustomSymbolicListener`. SPF provides a class that extends `PropertyListenerAdapter` called `SymbolicListener`. It collects the path conditions and return transformations of the execution paths of the method under check. The class `CustomSymbolicListener` is a modified version of `SymbolicListener` that collects, in addition to the return transformations, fields transformations and adds representations of data structure objects assigned to field in transformations.

  Two of the fields that `CustomSymbolicListener` have are: `methodsSymbolicSummaries`, which is a map that stores the symbolic summaries of the methods under check, and `transformedSymFields` which is a list of symbolic fields that the method under check has written to.

  The methods of `CustomSymbolicListener` that are relevant to the collection of fields and objects transformations are `executeInstruction` and `instructionExecuted`. The JPF Virtual Machine (VM) calls the `executeInstruction` methods of the registered listeners just prior to the execution of the current instruction.

  The method `executeInstruction` in `CustomSymbolicListener` checks if the current instruction is the first instruction in the current execution

path to write to a symbolic field. These instructions are `PUTFIELD` and `PUTSTATIC`. If that is the case, `executeInstruction` instantiates an object of type `TransformedSymField` and stores the field information and information that identifies the execution path that wrote to the field in that object. It then adds the instantiated object to the list `transformedSymFields`.

The JPF VM calls the method `instructionExecuted` after execution of the current instruction. `CustomSymbolicListener`'s implementation of `instructionExecuted` is responsible for the construction of the symbolic summaries of the execution paths and method under check. The method `instructionExecuted` checks if the last instruction executed was a return instruction. If so, it adds the return transformation, then iterates over `transformedSymFields` and constructs the transformations of the fields that the current execution path has written to. The method then constructs the path summary and adds it to method's summary.

- `TransformedSymField`. This class represents a symbolic field that an execution path has written to. The class has fields that specify the transformed field's symbol, type, and a reference to the owner object of the field. `TransformedSymField` also stores information that identifies the execution path that has written to the field. The class includes a method `getTransformationConstraint` that constructs the transformation expression specific to the field. `CustomSymbolicListener.instructionExecuted()` calls this method when it is iterating over the transformed fields and constructing the path's summary. `TransformedSymField` also provides the method `getConcreteObjectFieldsTransformations()` that adds transformations of dynamic data structures. It implements EncodeTransforms specified in Algorithm 1. The method `getTransformationConstraint()` calls `getConcreteObjectFieldsTransformations()` whenever the transformed field is assigned a concrete (non-symbolic) reference to an object on the heap.

- `SymbolicPathSummary`. This class represents the symbolic summary of an execution path. A `SymbolicPathSummary` object stores the path con-

dition and transformations of an execution path of the method under check. The class provides a method `completeTransformations()` that adds the identity expressions for the unchanged fields as described in Equation (4.4).

- `SymbolicMethodSummary`. A `SymbolicMethodSummary` object stores the symbolic summary of the method under check in the form of a list of SymbolicPathSummary objects.

Chapter 6

Evaluation

The goal of this chapter is to provide an assessment of HeapChecker's performance in terms of correctness and efficiency when checking heap-modifying programs.

Prior equivalence checking tools have primarily focused on perfecting methods for abstraction and refinement that accomplish a complete and precise analysis. The focus of HeapChecker, however, is to allow for the analysis of programs that operate on the heap. Specifically, programs that write or modify fields and parameters, instantiate objects on the heap, and receive input of parameterized type. HeapChecker intersects with previous checkers in providing equivalence check for programs with basic arithmetic operations, looping, and conditionals. In order to assess HeapChecker's efficiency on heap-modifying programs, we first evaluate HeapChecker's performance on this area of intersection with previous tools. This provides us with a baseline which we can then use to assess HeapChecker's performance on heap-modifying benchmarks. We summarize these goals in the following research questions:

- RQ1: How does the performance of HeapChecker compare to previous equivalence checkers on programs with basic arithmetic operations, loops, and conditionals?
- RQ2: Does HeapChecker correctly check programs with writes to fields, dynamic allocation of objects, and parameterized types?

- RQ3: How efficient is HeapChecker in checking programs with writes to fields, dynamic allocation of objects, and parameterized types?

6.1 Experimental Setup

To address RQ1, we evaluated HeapChecker on the benchmarks that Trostanetski et al. selected to evaluate their tool, ModDiff [33]. We refer to these benchmarks as the ModDiff benchmarks. Badihi et al. provide the implementation of their tool, ARDiff, along with two additional tools DSE [27] and IMP-S [3]. The implementations of the three tools are available online [2]. To provide a fair comparison, we execute these implementations along with HeapChecker on the ModDiff benchmarks using the same machine.

Given that the programs in the existing benchmarks do not contain the features HeapChecker focuses on, we created a new set of benchmarks To answer RQ2 and RQ3. We refer to these as HeapChecker benchmarks.

We ran all experiments on an x86_64 Ubuntu 18.04.5 LTS machine with four AMD EPYC 7351 16-Core processors and 472GB of memory. We built HeapChecker using OpenJDK 1.8.0_282. We compiled all subjects and ran all tools using Oracle JDK 1.8.0_201.

6.2 RQ1: How does the performance of HeapChecker compare to previous checkers on basic programs?

The ModDiff benchmarks comprise 28 benchmarks. Each benchmark contains two versions of a method. Of those, 16 pairs are equivalent and 12 pairs are non-equivalent. Figure 6.1 shows samples of these benchmarks.

We execute each tool on each benchmark 10 times. Figures 6.2, 6.3, 6.4, and 6.5 depict boxplots of the execution times of HeapChecker, ARDiff, DSE, and IMP-S on the equivalent pairs of the ModDiff benchmarks. Figures 6.6, 6.7, 6.8, and 6.9 depict the execution times of HeapChecker, ARDiff, DSE, and IMP-S on the non-equivalent pairs of the ModDiff benchmarks. Each

```
int snippet(int x) {
  if ((x >= 5) && (x < 7)) {
    int c = 0;
    for (int i = 1; i <= x; ++i)
      c += 5;
    return c;
  }
  return 0;
}
```

```
int snippet(int x) {
  if ((x >= 5) && (x < 7)) {
    int c = 0;
    for (int i = 1; i <= 5; ++i)
      c += x;
    return c;
  }
  return 0;
}
```

Two equivalent versions of the LoopMult5 benchmark.

```
int snippet(int x) {
  if ((x >= 9) && (x < 12)) {
    int c = 0;
    if (x < 0) {
      for (int i = 1; i <= 10; ++i)
        c += x;
    }
    return c;
  }
  return 0;
}
```

```
int snippet(int x) {
  if ((x >= 9) && (x < 12)) {
    int c = 0;
    if (x < 0) {
      for (int i = 1; i <= x; ++i)
        c += 10;
    }
    return c;
  }
  return 0;
}
```

Two equivalent versions of the LoopUnreach10 benchmark.

Figure 6.1: ModDiff benchmarks samples.

boxplot depicts the execution times of a tool on a benchmark.

Figure 6.2 shows that for equivalent pairs, except LoopMult2 and UnchLoop, HeapChecker's execution times range from 2.1 to 2.4 seconds. For LoopMult2 HeapChecker's execution times range is between 3.0 and 3.6 seconds, while for UnchLoop it is 2.8 and 3.2 seconds. For ARDiff, execution times on equivalent pairs (Figure 6.3) vary from a minimum of 3 seconds for LoopSub and Sub benchmarks, up to 13 seconds for the Comp benchmark. ARDiff's times for LoopMult2 range between 7.7 and 8 seconds. For UnchLoop, ARDiff's execution times range between 6.2 and 6.4 seconds. DSE's execution times on equivalent pairs (Figure 6.4) range from 3 seconds for the Sub benchmark up to 8.4 seconds for the LoopMult20 benchmark. For LoopMult2, DSE's times range from 4.7 to 5.0 seconds, while for UnchLoop DSE's execution times range from 3.0 to 3.2 seconds. IMP-S' execution times on equivalent pairs (Figure 6.5) vary from a minimum of 3 seconds for the Sub benchmark and a maximum of 8 seconds for the LoopMult20 benchmark. The LoopMult2

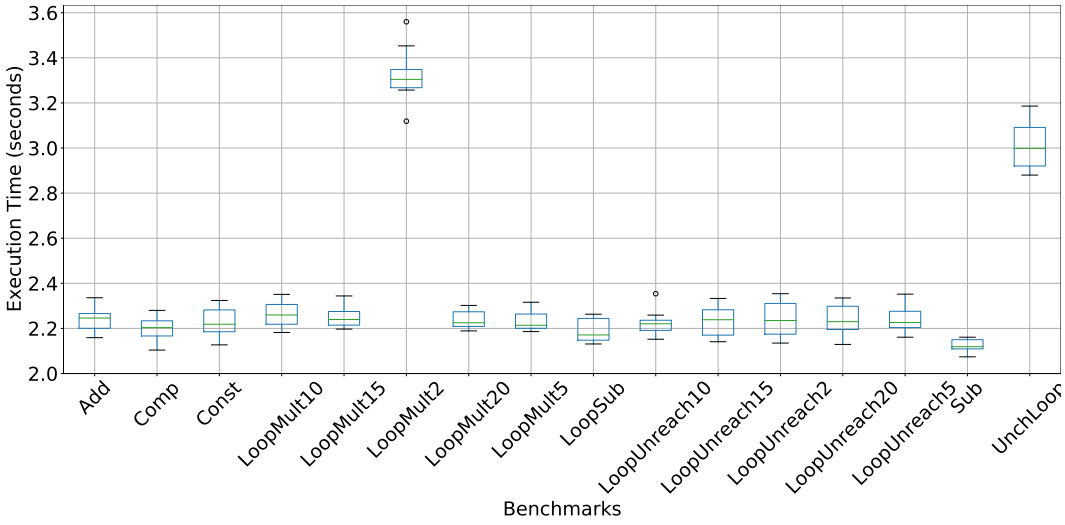


Figure 6.2: HeapChecker's execution times on equivalent pairs of ModDiff benchmarks.

benchmark ranges from 4.7 to 4.9 seconds, while the UnchLoop ranges from 4.7 to 5 seconds.

For the equivalent pair of the UnchLoop benchmark, HeapChecker's execution times (2.8-3.2) are similar to DSE's (3.0-3.2), and shorter the other tools. For the rest of the equivalent pairs benchmarks, HeapChecker's execution times are shorter than the execution times of the other tools. For the non-equivalent pairs (Figures 6.6, 6.7, 6.8, and 6.9), HeapChecker's execution times are shorter than the other tools.

The shorter execution times of HeapChecker may be due to the abstraction employed by DSE, the abstraction-refinement iterations ARDiff performs, and the static analysis in IMP-S. Since HeapChecker does not implement any abstraction, refinement, or prior static analysis, it exhibits shorter execution times in benchmarks that are sufficiently simple as in the case with ModDiff benchmarks. The abstraction and static analysis techniques pay off in programs with complex operations and constraints where abstracting parts containing such operations does not affect the equivalence checking decision. HeapChecker, however, does not address this aspect.

HeapChecker's execution times range from 2.1 to 3.1 and are shorter than other tools times (DSE, ARDiff, and IMP-S) on most of the ModDiff benchmarks.

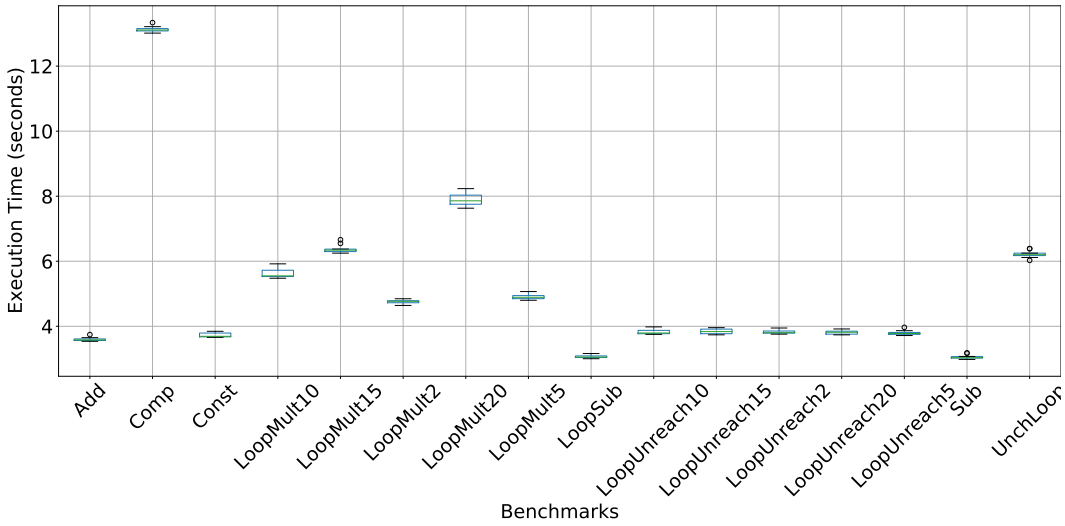


Figure 6.3: ARDiff's execution times on equivalent pairs of ModDiff benchmarks.

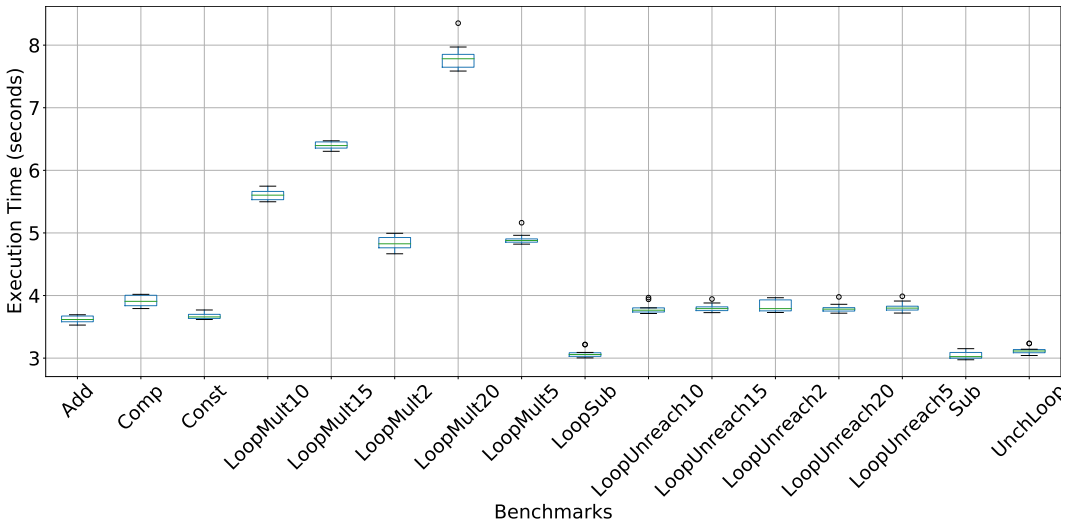


Figure 6.4: DSE's execution times on equivalent pairs of ModDiff benchmarks.

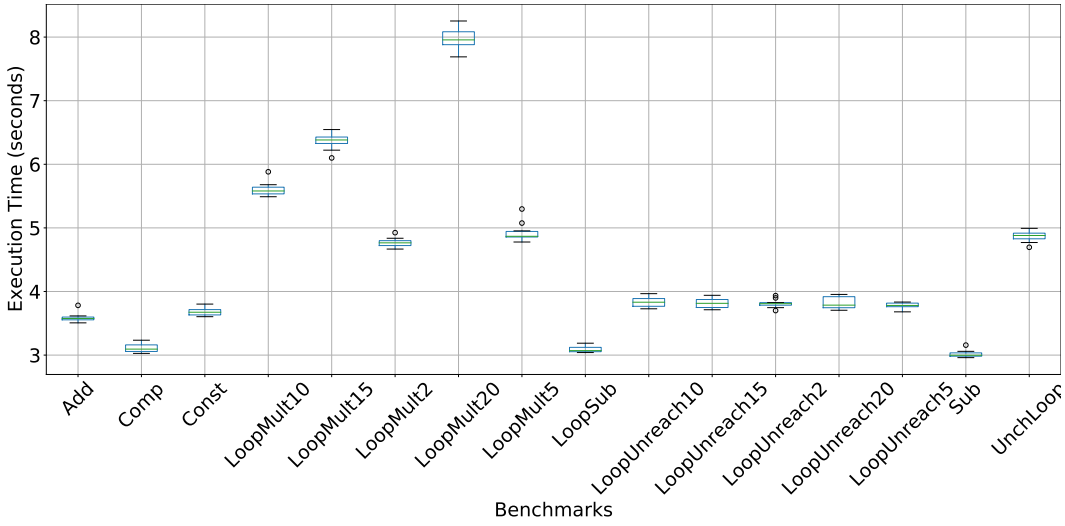


Figure 6.5: IMP-S's execution times on equivalent pairs of ModDiff benchmarks.

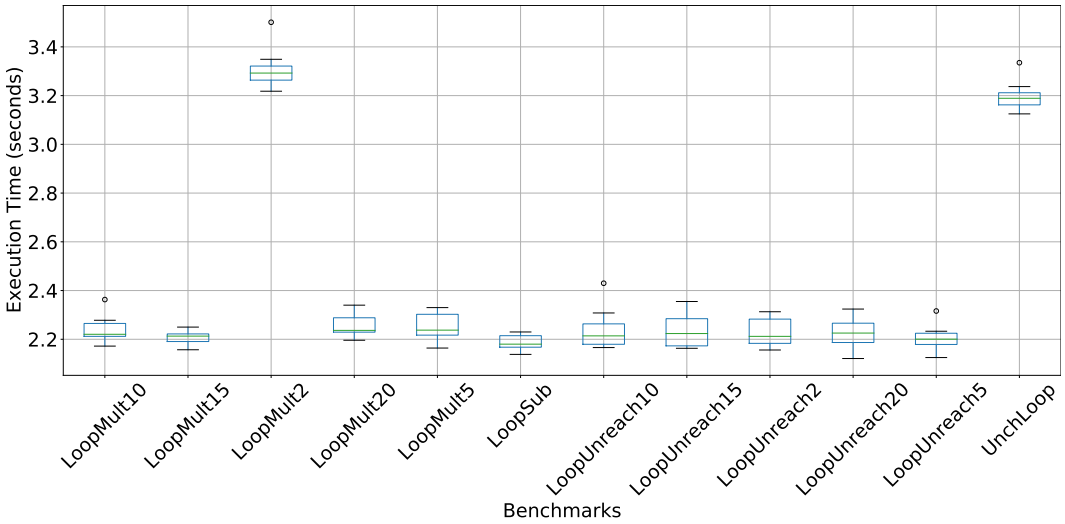


Figure 6.6: HeapChecker's execution times on non-equivalent pairs of ModDiff benchmarks.

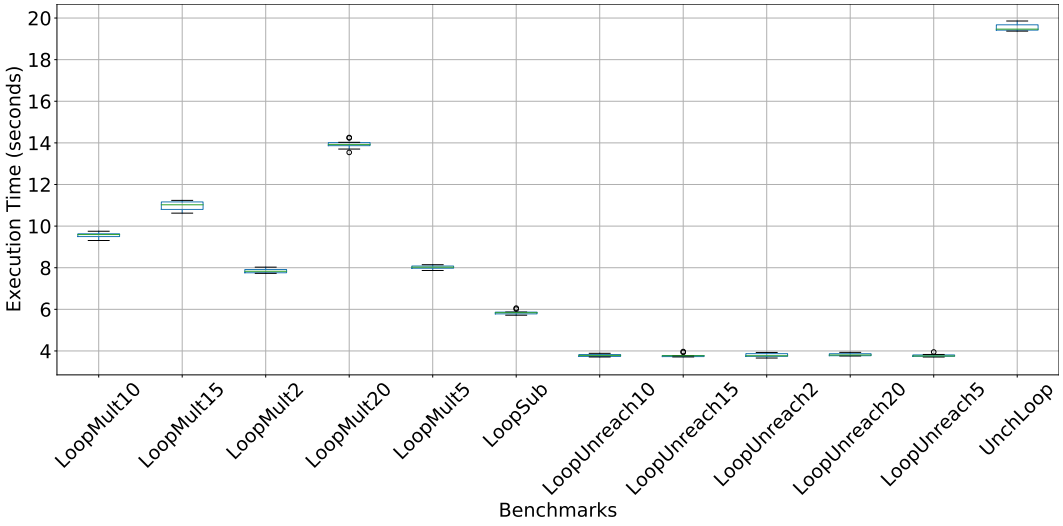


Figure 6.7: ARDiff's execution times on non-equivalent pairs of ModDiff benchmarks.

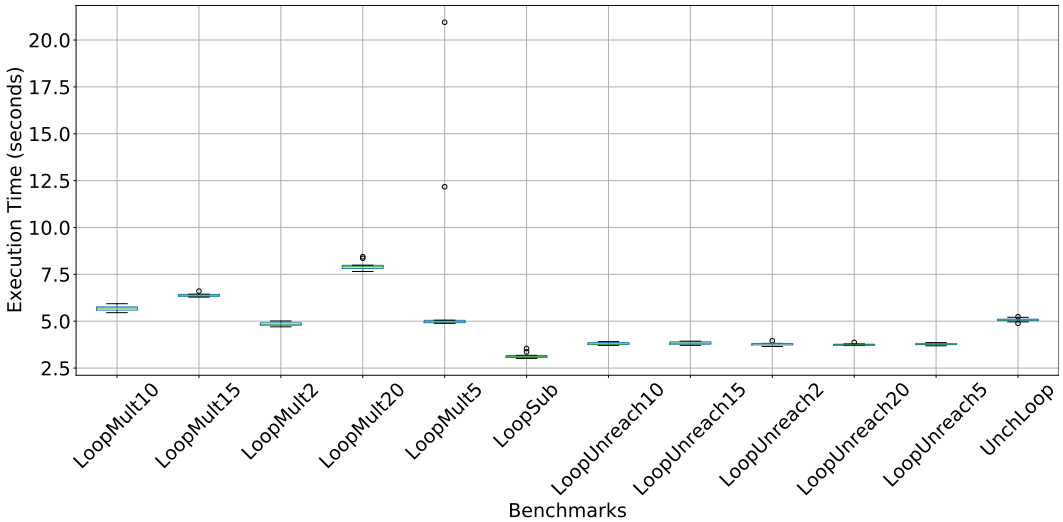


Figure 6.8: DSE's execution times on non-equivalent pairs of ModDiff benchmarks.

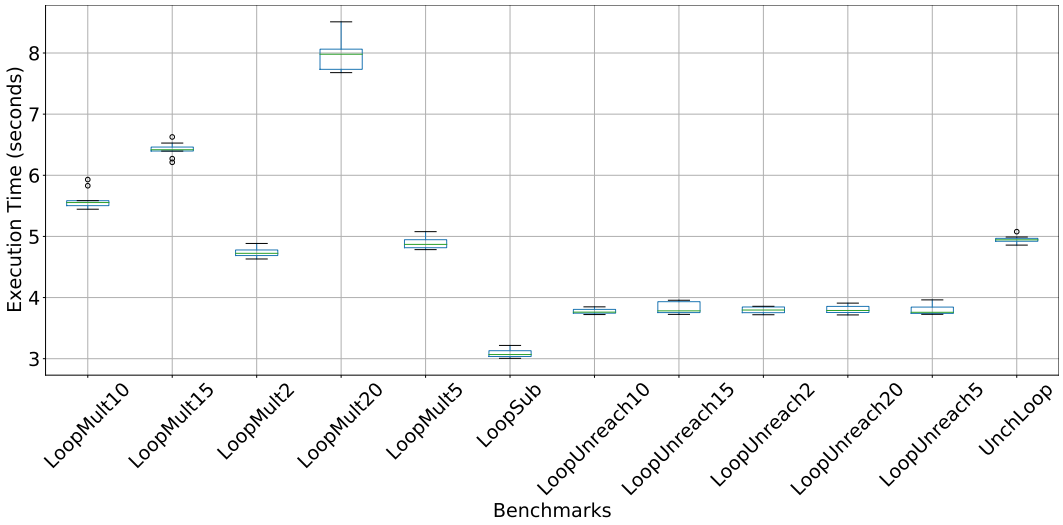


Figure 6.9: IMP-S's execution times on non-equivalent pairs of ModDiff benchmarks.

6.3 RQ2: Does HeapChecker correctly check heap programs?

6.3.1 HeapChecker Benchmarks

The HeapChecker benchmarks consist of 15 benchmarks. Each benchmark contains two pairs of methods: one equivalent pair and one non-equivalent pair, thus totalling 30 pairs of methods. The benchmarks exhibit the features that we want to assess HeapChecker's performance on: writes to fields, dynamic objects allocation, and parameters and fields of parameterized types. The details of these benchmarks are provided in Appendix A.

Table 6.1 shows the results of running HeapChecker on the HeapChecker benchmarks. HeapChecker succeeds in correctly deciding the equivalence and non-equivalence of all benchmarks, except the equivalent pair of Alias2, shown in Figure 6.10.

The aliasing choice in lazy initialization assigns the field (or parameter) being initialized the symbol of a previously initialized field of the same type, that has been assigned a new object. The aliasing initialization option is necessary for HeapChecker to explore execution paths that depends on two fields being aliases.

Table 6.1: Correctness results of running HeapChecker on the HeapChecker benchmarks.

Benchmarks	Equivalent Benchmarks	Non-equivalent Benchmarks
Lazy	✓	✓
Dynamic	✓	✓
Generic	✓	✓
Alias1	✓	✓
Alias2	✗	✓
BuildList5	✓	✓
BuildList10	✓	✓
BuildList20	✓	✓
SetList5	✓	✓
SetList10	✓	✓
SetList20	✓	✓
BuildTree5	✓	✓
BuildTree10	✓	✓
SetTree1	✓	✓
SetTree2	✓	✓

```
149 void snippet(Tree t, int x) {
150   if (t != null) {
151     if ((t.left == t.right) && (t
          .left != null))

152       t.left.content = x;
153   }
154 }
```

(a) Version 1: `t.left` is read first.

```
155 void snippet(Tree t, int x) {
156   if (t != null) {
157     if ((t.right == t.left) && (t
          .left != null))

158       t.left.content = x;
159   }
160 }
```

(b) Version 2: `t.right` is read first.

Figure 6.10: Equivalent pair of Alias2 benchmark.

For execution paths that HeapChecker takes only when aliasing between two fields exists (as in Alias2 in Figure 6.10), the resulting transformations reflect this aliasing. Specifically, the transformation encodes either of the two aliased symbols, but not both of them. A consequence of this is that for two execution paths that are equivalent, their resulting summaries may not reflect that because the transformations for one summary contain only the transformations of one field, while the other summary contains the transformations of the other field that aliases with the first one. The `snippet` methods of Alias2 in Figure 6.10 are identical, save for the change in the order of the operands of the comparison expressions in Line 151 and Line 157 in Figure 6.10. This change translates to change in the order of how the two parameters are accessed. In the first version, in Line 151, `t.left` is accessed first and its initialization options are either null or a new object, since no other object of the other type has been accessed. After `t.right` is accessed, it is initialization options are null, the same object assigned to `t.left` (since it has already been initialized), or a new object.

The path summary corresponding to the aliasing case for version 1 is:

```
t.right = t.left && t.left.content_out = x
```

The reverse happens in the second version, where codet.right is accessed first, resulting in the following summary for version 2:

```
t.left = t.right && t.right.content_out = x
```

The resulting summaries are non-equivalent. For equivalence to work in this case, additional transformations need to be added for each aliasing case:

```
t.right = t.left && t.left.content_out = x && t.right.content_out = x
t.left = t.right && t.right.content_out = x && t.left.content_out = x
```

HeapChecker does not currently add transformations for aliasing possibilities, so it fails to correctly decide the equivalence of the pair of Alias2. A potential remedy to this problem, is for each aliasing expression (e.g., `t.right = t.left`) that HeapChecker encounters, HeapChecker checks if there is a transformation involving any fields of the aliasing symbols (e.g., `t.left.content_out = x`). If yes, then HeapChecker adds a corresponding transformation for the field of the other symbol (e.g., `t.right.content_out =`

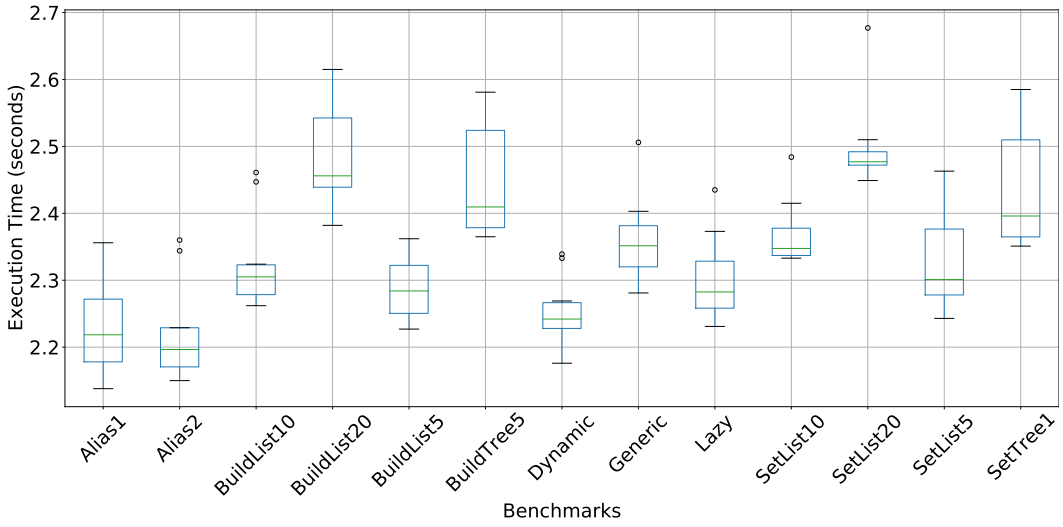


Figure 6.11: HeapChecker's execution times on equivalent pairs of HeapChecker benchmarks.

x).

> HeapChecker correctly checks programs with writes to fields, dynamic objects allocation, and parameters and fields of parameterized types, with no explicit alias checks. HeapChecker, however, fails to correctly check programs with explicit alias checks.

6.4 RQ3: How efficient is HeapChecker in checking programs with writes to fields, dynamic allocation of objects, and parameterized types?

Figure 6.11, Figure 6.12, and Figure 6.13 show the execution times results for running HeapChecker on the HeapChecker benchmarks. To obtain those results, we ran HeapChecker on each benchmark 10 times, with the aliasing option for lazy initialization disabled. For most benchmarks, the execution time remains below 3 seconds (Figure 6.11 and Figure 6.12), similar to the range of the execution times on the ModDiff benchmarks (Figure 6.2 and Figure 6.6). The notable exceptions to this are benchmarks BuildTree10, and SetTree2 (Figure 6.13).

For the equivalent pair of BuildTree10 (Figure 6.14), the execution time for most runs ranges between 30 and 40 seconds, and exceeds 120 seconds in

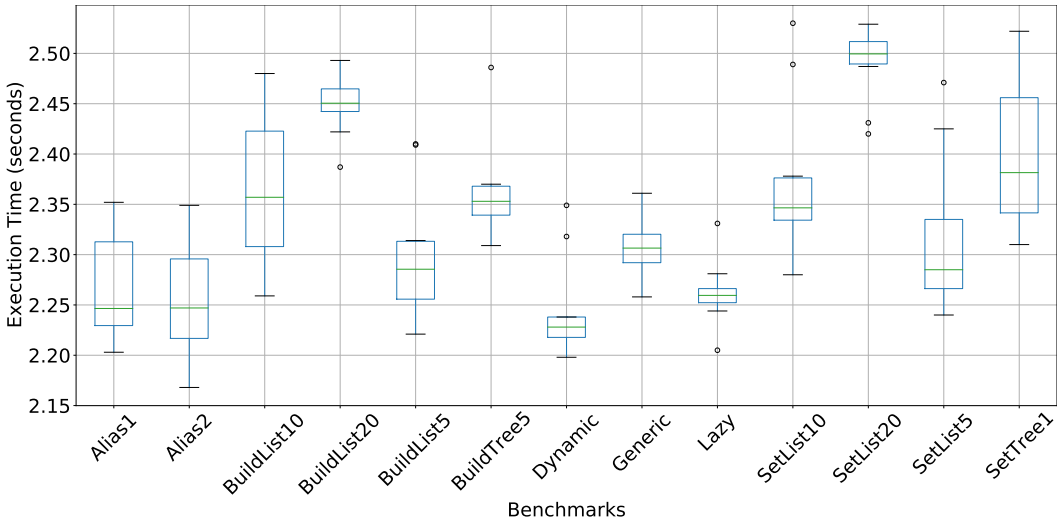
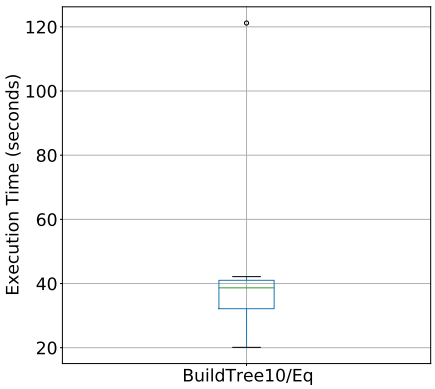


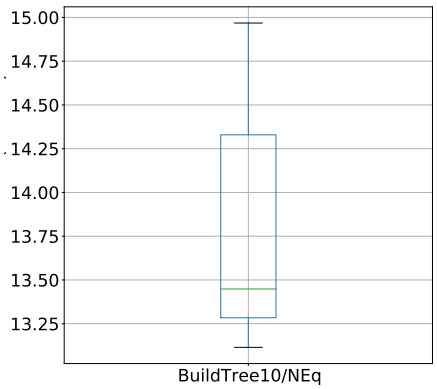
Figure 6.12: HeapChecker's execution times on non-equivalent pairs of HeapChecker benchmarks.

one instance (Figure 6.13a). The reason for this is that BuildTree10 is an exponential-time method with an upper-bound of 10, resulting in a very large number of execution paths. Similarly, we notice large execution times (12-15 seconds) for the non-equivalent pair of BuildTree10 (Figure 6.15), but less than the equivalent pair (Figure 6.13b). The reason for this is that version 1 of the `snippet` method contains two recursive calls (Line 192), resulting in an exponential time complexity, while for version 2, the method contains a single recursive call (Line 204), thus resulting in linear complexity and a small execution time, and also being the reason for the non-equivalence between the two versions.

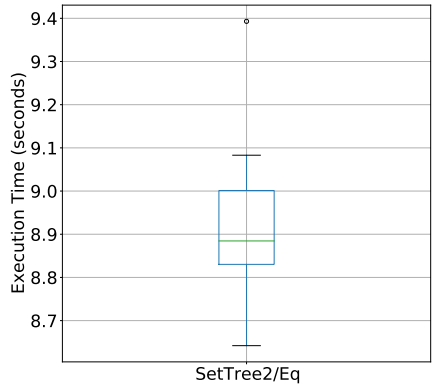
Despite having only an upper bound of 2, SetTree2 (Figure 6.16) execution times (Figure 6.13c) are higher than BuildTree5, despite having a smaller upper bound. The reason for this is the execution paths that lazy initialization generates in the case of SetTree2. Since the method `snippet` of SetTree2 takes as an input a reference type parameter, HeapChecker explores two possible execution paths (not counting aliasing possibilities) for each field accessed for the first time. Since two new fields get accessed (`t.left` and `t.right` in Line 214 and Line 215) in each call, then 4 new execution paths result from each call, to account for all initialization possibilities of the accessed fields.



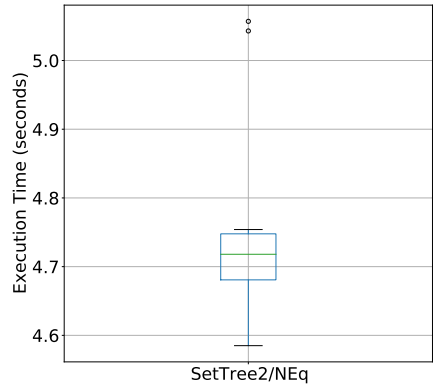
(a) HeapChecker's execution times on equivalent pair of the BuildTree10 benchmark.



(b) HeapChecker's execution times on non-equivalent pair of the BuildTree10 benchmark.



(c) HeapChecker's execution times on equivalent pair of the SetTree2 benchmark.



(d) HeapChecker's execution times on non-equivalent pair of the SetTree2 benchmark.

Figure 6.13: HeapChecker's execution times on the BuildTree10 and SetTree2 benchmark.

```
165 public Tree snippet(int x) {
166   if ((x >= (-1)) && (x < 9)) {
167     if (x <= 0) {
168       return new Tree(x, null,
              null);
169     } else {
170       return new Tree(x, snippet(
              x - 1), snippet(x - 1))
              ;
171     }
172   }
173   return null;
174 }
```

```
175 public Tree snippet(int x) {
176   if ((x >= (-1)) && (x < 9)) {
177     if (x == 0) {
178       return new Tree(0, null,
              null);
179     } else if (x <= 0) {
180       return new Tree(x, null,
              null);
181     } else {
182       return new Tree(x, snippet(
              x - 1), snippet(x - 1))
              ;
183     }
184   }
185   return null;
186 }
```

Figure 6.14: Equivalent pair of BuildTree10 benchmark.

```
187 Tree snippet(int x) {
188   if ((x >= (-1)) && (x < 9)) {
189     if (x <= 0) {
190       return new Tree(x, null,
              null);
191     } else {
192       return new Tree(x, snippet(
              x - 1), snippet(x - 1))
              ;

193     }
194   }
195   return null;
196 }
```

(a) Version 1: Two recursive calls.

```
197 Tree snippet(int x) {
198   if ((x >= (-1)) && (x < 9)) {
199     if (x == 0) {
200       return new Tree(0, null,
              null);
201     } else if (x <= 0) {
202       return new Tree(x, null,
              null);
203     } else {
204       Tree t = snippet(x - 1);
205       return new Tree(x, t, t);
206     }
207   }
208   return null;
209 }
```

(b) Version 2: A single recursive call.

Figure 6.15: Non-equivalent pair of BuildTree10 benchmark.

```
210 void snippet(Tree t, int x) {
211   if ((x >= 0) && (x < 3)) {
212     if (t != null) {
213       t.content = x;
214       snippet(t.left, x - 1);
215       snippet(t.right, x - 1);
216     }
217   }
218 }
```

```
220 void snippet(Tree t, int x) {
221   if ((x >= 0) && (x < 3)) {
222     if (t != null) {
223       snippet(t.right, x - 1);
224       snippet(t.left, x - 1);
225       t.content = x;
226     }
227   }
228 }
```

Figure 6.16: Equivalent pair of SetTree2 benchmark.

This results in the number of execution paths growing rapidly with higher upper bounds. The BuildTree benchmarks do not take reference type input, so they do not exhibit similarly large execution times.

HeapChecker efficiently checks programs with dynamic allocation of objects, parameterized types, and non-recursive programs that read and write to reference type input. HeapChecker, however, is inefficient when checking recursive programs that read and write to reference type input.

Chapter 7

Conclusion

This thesis details the design, implementation, and evaluation of HeapChecker. HeapChecker is an equivalence checker for Java programs that contain heap-related features: writing or modifying instance fields and parameters, allocating objects on the heap, and dealing with input of paramaterized types. HeapChecker builds on and leverages the capabilities of the symbolic executor SPF. Specifically, it relies on SPF to explore the execution paths of the programs under check that result from branching statements and the initialization options of reference type input. HeapChecker builds symbolic summaries during the symbolic execution and verifies whether the summaries are equivalent or not using a satisfiability solver such as Z3.

The evaluation shows that HeapChecker is capable of correctly checking the equivalence of programs containing heap-related features, with the exception of programs containing explicit alias checks. The performance of HeapChecker remains within similar ranges to previous tools, with the exception of cases that involve recursive methods taking reference type input.

The following are ideas for improving HeapChecker:

- Including transformations for fields of aliasing references to correctly check programs with execution paths that depend on alias checks.
- Adding support for methods symbolic arrays, based on SPF support for symbolic arrays [13].
- Including summaries for exceptional execution paths.

- Integrating HeapChecker's approach to heap-related features with abstraction and refinement techniques to help alleviate HeapChecker's scalability issues and to deal with programs containing complex constraints.

References

[1] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 367–381.

[2] (). Ardiff_equiv_checking, [Online]. Available: `https://github.com/shrBadihi/ARDiff%5C_Equiv%5C_Checking` (visited on 03/31/2021).

[3] J. Backes, S. Person, N. Rungta, and O. Tkachuk, "Regression verification using impact summaries," in *Model Checking Software*, 2013, pp. 99–116.

[4] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin, "Ardiff: Scaling program equivalence checking via iterative abstraction and refinement of common code," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 13–24. DOI: `10.1145/3368089.3409757`.

[5] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, May 2018. DOI: `10.1145/3182657`.

[6] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability*. 2009, vol. 185.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, 2011, pp. 463–469.

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 209–224.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012. DOI: `10.1145/2110356.2110358`.

[10] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168–176.

[11] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, 2008, pp. 337–340.

[12] X. Deng, J. Lee, and Robby, "Efficient and formal generalized symbolic execution," *Automated Software Engineering*, pp. 233–301, 2012.

[13] A. Fromherz, K. S. Luckow, and C. S. Păsăreanu, "Symbolic arrays in symbolic pathfinder," *SIGSOFT Softw. Eng. Notes*, pp. 1–5, 2017. DOI: 10.1145/3011286.3011296.

[14] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036.

[15] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, 2008, pp. 151–166.

[16] B. Godlin and O. Strichman, "Inference rules for proving the equivalence of recursive procedures," *Acta Informatica*, vol. 45, no. 6, pp. 403–439, 2008.

[17] ——, "Regression verification," in *Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 466–471. DOI: 10.1145/1629911.1630034.

[18] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2015). The java® language specification, [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.10.2 (visited on 03/31/2021).

[19] (). Java pathfinder, [Online]. Available: https://github.com/javapathfinder/jpf-core/wiki (visited on 03/31/2021).

[20] S. Khurshid, C. S. PĂsĂreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2003, pp. 553–568.

[21] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, pp. 385–394, Jul. 1976. DOI: 10.1145/360248.360252.

[22] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman, "Jdart: A dynamic symbolic analysis framework," in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*, 2016, pp. 442–459. DOI: 10.1007/978-3-662-49674-9_26.

[23] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples, "Empirical study towards a leading indicator for cost of formal software verification," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 722–732. DOI: 10.1109/ICSE.2015.85.

[24] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, pp. 15–26. DOI: 10.1145/1390630.1390635.

[25] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: Symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, 2010, pp. 179–180. DOI: 10.1145/1858996.1859035.

[26] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. DOI: 10.1002/spe.2346. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document.

[27] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pundefinedsundefinedreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 226–237. DOI: 10.1145/1453101.1453131.

[28] L. H. Pham, Q. L. Le, Q.-S. Phan, J. Sun, and S. Qin, "Enhancing symbolic execution of heap-based programs with separation logic for test input generation," in *Automated Technology for Verification and Analysis*, 2019, pp. 209–227.

[29] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[31] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, 2008, pp. 1–25.

[32] (). Symbolic pathfinder, [Online]. Available: https://github.com/SymbolicPathFinder/jpf-symbc/wiki (visited on 03/31/2021).

[33] A. Trostanetski, O. Grumberg, and D. Kroening, "Modular demand-driven analysis of semantic difference for program versions," in *Static Analysis*, 2017, pp. 405–427.

[34] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, pp. 203–232, 2003. DOI: 10.1023/A:1022920129859.

Appendix A

HeapChecker Benchmarks

The HeapChecker benchmarks consist of 15 benchmarks. Each benchmark contains two pairs of methods: one equivalent pair and one non-equivalent pair, totalling 30 pairs of methods. The benchmarks can be divided into two categories: constant-time methods and iterative/recursive methods. The benchmarks methods contain writes to fields and parameters, dynamic objects allocation, and reads and writes to parameters and fields of parameterized types. The details of the benchmarks are as follows, where the target method in each benchmark is `snippet`:

- Lazy (Figure A.1 and Figure A.2): reads a fixed size linked list of integers and sets all its elements to the same value.
- Dynamic (Figure A.3 and Figure A.4): reads an integer and stores it in a fixed size dynamically allocated tree.
- Generic (Figure A.5 and Figure A.6): Same as Lazy, but the type of the linked list is an invocation of a generic type.
- Alias1 (Figure A.7 and Figure A.8): reads a binary tree, assigns its right node to its left node, then assigns an integer to the left or right node.
- Alias2 (Figure A.9 and Figure A.10): reads a binary tree and then assigns an integer to its left node only if both left and right reference the same object.

```
229 void snippet(LList list, int x) {
230   for (int i = 0; i < 10; i++) {
231     if (list == null)
232       return;
233     list.content = x;
234     list = list.next;
235   }
236 }
```

```
237 void snippet(LList list, int x) {
238   aux(list, x, 0);
239 }
240
241 void aux(LList list, int x, int length) {
242   if ((list == null) || (length == 10)) {
243     return;
244   } else {
245     list.content = x;
246     aux(list.next, x, length + 1);
247   }
248 }
```

Figure A.1: Equivalent pair of Lazy benchmark.

- BuildList5, BuildList10 (Figure A.11 and Figure A.12), and BuildList20: dynamically builds a linked list of variable size, with upper length bounds of 5, 10, and 20.
- SetList5 (Figure A.13 and Figure A.14), SetList10, and SetList20: writes to the element of a linked list of variable size, with upper length bounds of 5, 10, and 20.
- BuildTree5 (Figure A.15 and Figure A.16) and BuildTree10: dynamically builds a binary tree of variable size, with upper length bounds of 5 and 10.
- SetTree1 and SetTree2 (Figure A.17 and Figure A.18): writes to the element of a binary of variable size, with upper height bounds of 1 and 2.

```
249 void snippet(LList list, int x) {
250   for (int i = 0; i < 10; i++) {
251     if (list == null)
252       return;
253     list.content = x;
254     list = list.next;
255   }
256 }
```

```
257 void snippet(LList list, int x) {
258   aux(list, x, 0);
259 }
260
261 void aux(LList list, int x, int length) {
262   if ((list == null) || (length == 10)) {
263     return;
264   } else {
265     list.content = x;
266     aux(list, x, length + 1);
267   }
268 }
```

Figure A.2: Non-equivalent pair of Lazy benchmark.

```
269 Tree snippet(int x) {
270   if (x == 0) {
271     return new Tree(0, null, null);
272   } else if (x < 0) {
273     return new Tree(x, null, null);
274   } else {
275     Tree t1 = new Tree(x - 2, null, null);
276     Tree t2 = new Tree(x - 1, null, null);
277     return new Tree(x, t2, t1);
278   }
279 }
```

```
280 Tree snippet(int x) {
281   if (x <= 0) {
282     return new Tree(x, null, null);
283   } else {
284     Tree t1 = new Tree(x - 1, null, null);
285     Tree t2 = new Tree(x - 2, null, null);
286     return new Tree(x, t1, t2);
287   }
288 }
```

Figure A.3: Equivalent pair of Dynamic benchmark.

```
289 Tree snippet(int x) {
290   if (x == 0) {
291     return new Tree(0, null, null
          );
292   } else if (x < 0) {
293     return new Tree(x, null, null
          );
294   } else {
295     Tree t1 = new Tree(x - 2,
          null, null);
296     Tree t2 = new Tree(x - 1,
          null, null);
297     return new Tree(x, t2, t1);
298   }
299 }
```

```
300 Tree snippet(int x) {
301   if (x == 0) {
302     return new Tree(0, null, null
          );
303   } else if (x < 0) {
304     return new Tree(x, null, null
          );
305   } else {
306     Tree t1 = new Tree(x - 1,
          null, null);
307     Tree t2 = new Tree(x - 1,
          null, null);
308     return new Tree(x, t1, t1);
309   }
310 }
```

Figure A.4: Non-equivalent pair of Dynamic benchmark.

```
311 void snippet(LList<Integer> list
      , int x) {
312   for (int i = 0; i < 10; i++) {
313     if (list == null)
314       return;
315     list.content = x;
316     list = list.next;
317   }
318 }
```

```
319 void snippet(LList<Integer> list
      , int x) {
320   aux(list, x, 0);
321 }
322
323 public static void aux(LList<
      Integer> list, int x, int
      length) {
324   if ((list == null) || (length
        == 10)) {
325     return;
326   } else {
327     list.content = x;
328     aux(list.next, x, length + 1)
          ;
329   }
330 }
```

Figure A.5: Equivalent pair of Generic benchmark.

```
void snippet(LList<Integer> list
    , int x) {
  for (int i = 0; i < 10; i++) {
    if (list == null)
      return;
    list.content = x;
    list = list.next;
  }
}
```

```
void snippet(LList<Integer> list
    , int x) {
  aux(list, x, 0);
}

public static void aux(LList<
    Integer> list, int x, int
    length) {
  if ((list == null) || (length
      == 10)) {
    return;
  } else {
    list.content = x;
    aux(list, x, length + 1);
  }
}
```

Figure A.6: Non-equivalent pair of Generic benchmark.

```
void snippet(Tree t, int x) {
  if (t != null) {
    t.left = t.right;
    if (t.left != null)
      t.left.content = x;
  }
}
```

```
void snippet(Tree t, int x) {
  if (t != null) {
    t.left = t.right;
    if (t.right != null)
      t.right.content = x;
  }
}
```

Figure A.7: Equivalent pair of Alias1 benchmark.

```
void snippet(Tree t, int x) {
  if (t != null) {
    t.left = t.right;
    if (t.left != null)
      t.left.content = x;
  }
}
```

```
void snippet(Tree t, int x) {
  if (t != null) {
    t.right = t.left;
    if (t.right != null)
      t.right.content = x;
  }
}
```

Figure A.8: Non-equivalent pair of Alias1 benchmark.

```
void snippet(Tree t, int x) {
  if (t != null) {
    if ((t.left == t.right) && (t
        .left != null))
      t.left.content = x;
  }
}
```

```
void snippet(Tree t, int x) {
  if (t != null) {
    if ((t.right == t.left) && (t
        .left != null))
      t.left.content = x;
  }
}
```

Figure A.9: Equivalent pair of Alias2 benchmark.

```
391 void snippet(Tree t, int x) {
392   if (t != null) {
393     if ((t.left == t.right) && (t
          .left != null))
394       t.left.content = x;
395   }
396 }
```

```
397 void snippet(Tree t, int x) {
398   if (t != null) {
399     if ((t.right == t.left) && (t
          .left != null))
400       return;
401     else if (t.left != null)
402       t.left.content = x;
403   }
404 }
```

Figure A.10: Non-equivalent pair of Alias2 benchmark.

```
405 LList snippet(int num, int x) {
406   if ((x > 0) && (x < 11)) {
407     LList head = new LList(num,
          null);
408     LList list = head;
409     for (int i = 1; i < x; i++) {
410       list.next = new LList(num,
            null);
411       list = list.next;
412     }
413     return head;
414   }
415   return null;
416 }
```

```
417 LList snippet(int num, int x) {
418   if ((x > 0) && (x < 11)) {
419     LList nextNode = new LList(num, null
          );
420     LList head = nextNode;
421     for (int i = x - 1; i > 0; i--) {
422       head = new LList(num, nextNode);
423       nextNode = head;
424     }
425     return head;
426   }
427   return null;
428 }
```

Figure A.11: Equivalent pair of BuildList10 benchmark.

```
429 LList snippet(int num, int x) {
430   if ((x > 0) && (x < 11)) {
431     LList head = new LList(num,
          null);
432     LList list = head;
433     for (int i = 1; i < x; i++) {
434       list.next = new LList(num,
            null);
435       list = list.next;
436     }
437     return head;
438   }
439   return null;
440 }
```

```
441 LList snippet(int num, int x) {
442   if ((x > 0) && (x < 11)) {
443     LList nextNode = new LList(num, null
          );
444     LList head = nextNode;
445     for (int i = x; i > 0; i--) {
446       head = new LList(num, nextNode);
447       nextNode = head;
448     }
449     return head;
450   }
451   return null;
452 }
```

Figure A.12: Non-equivalent pair of BuildList10 benchmark.

```
453 void snippet(LList list, int num
      , int x) {
454   if ((x >= 0) && (x < 6)) {
455     for (int i = 0; i < x; i++) {
456       if (list == null) return;
457       list.content = num;
458       list = list.next;
459     }
460   }
461 }
```

```
462 void snippet(LList list, int num
      , int x) {
463   if ((x >= 0) && (x < 6)) {
464     if (list != null) {
465       for (int i = 0; i < x; i++)
            {
466         LList elem = list.get(i);
467         if (elem == null) return;
468         elem.content = num;
469       }
470     }
471   }
472 }
473
474 public LList get(int x) {
475   LList current = this;
476   for (int i = 0; i < x; i++) {
477     if (current == null) return
          null;
478     current = current.next;
479   }
480   return current;
481 }
```

Figure A.13: Equivalent pair of SetList5 benchmark.

```
482 void snippet(LList list, int num
      , int x) {
483   if ((x >= 0) && (x < 6)) {
484     for (int i = 0; i < x; i++) {
485       if (list == null) return;
486       list.content = num;
487       list = list.next;
488     }
489   }
490 }
```

```
491 void snippet(LList list, int num
      , int x) {
492   if ((x >= 0) && (x < 6)) {
493     if (list != null) {
494       for (int i = 0; i < x; i++)
            {
495         LList elem = list.get(i);
496         if (elem == null) return;
497         elem.content = num;
498       }
499     }
500   }
501 }
502
503 public LList get(int x) {
504   LList current = this;
505   for (int i = 0; i <= x; i++) {
506     if (current == null) return
          null;
507     current = current.next;
508   }
509   return current;
510 }
```

Figure A.14: Non-equivalent pair of SetList5 benchmark.

```
511 public Tree snippet(int x) {
512   if ((x >= (-1)) && (x < 9)) {
513     if (x <= 0) {
514       return new Tree(x, null,
              null);
515     } else {
516       return new Tree(x, snippet(
              x - 1), snippet(x - 1))
              ;
517     }
518   }
519   return null;
520 }
```

```
521 public Tree snippet(int x) {
522   if ((x >= (-1)) && (x < 9)) {
523     if (x == 0) {
524       return new Tree(0, null,
              null);
525     } else if (x <= 0) {
526       return new Tree(x, null,
              null);
527     } else {
528       return new Tree(x, snippet(
              x - 1), snippet(x - 1))
              ;
529     }
530   }
531   return null;
532 }
```

Figure A.15: Equivalent pair of BuildTree10 benchmark.

```
533 public Tree snippet(int x) {
534   if ((x >= (-1)) && (x < 9)) {
535     if (x <= 0) {
536       return new Tree(x, null,
              null);
537     } else {
538       return new Tree(x, snippet(
              x - 1), snippet(x - 1))
              ;
539     }
540   }
541   return null;
542 }
```

```
543 public Tree snippet(int x) {
544   if ((x >= (-1)) && (x < 9)) {
545     if (x == 0) {
546       return new Tree(0, null,
              null);
547     } else if (x <= 0) {
548       return new Tree(x, null,
              null);
549     } else {
550       Tree t = snippet(x - 1);
551       return new Tree(x, t, t);
552     }
553   }
554   return null;
555 }
```

Figure A.16: Non-equivalent pair of BuildTree10 benchmark.

```
556 void snippet(Tree t, int x) {
557   if ((x >= 0) && (x < 3)) {
558     if (t != null) {
559       t.content = x;
560       snippet(t.left, x - 1);
561       snippet(t.right, x - 1);
562     }
563   }
564 }
```

```
566 void snippet(Tree t, int x) {
567   if ((x >= 0) && (x < 3)) {
568     if (t != null) {
569       snippet(t.right, x - 1);
570       snippet(t.left, x - 1);
571       t.content = x;
572     }
573   }
574 }
```

Figure A.17: Equivalent pair of SetTree2 benchmark.

```
575 void snippet(Tree t, int x) {
576   if ((x >= 0) && (x < 3)) {
577     if (t != null) {
578       t.content = x;
579       snippet(t.left, x - 1);
580       snippet(t.right, x - 1);
581     }
582   }
583 }
```

```
585 void snippet(Tree t, int x) {
586   if ((x >= 0) && (x < 3)) {
587     if (t != null) {
588       snippet(t.right, x - 2);
589       snippet(t.left, x - 1);
590       t.content = x;
591     }
592   }
593 }
```

Figure A.18: Non-equivalent pair of SetTree2 benchmark.