CONTRACT-BASED VERIFICATION AND TEST CASE GENERATION FOR OPEN SYSTEMS

by

XIANGHUA DENG

B.S., Xi'an Jiaotong University, China, 1993
M.S., Kansas State University, 2001

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2007

Abstract

Current practices in software development heavily emphasize the development of reusable and modular software, which allow software components to be developed and maintained independently. While a component-oriented approach offers a number of benefits, it presents several quality assurance challenges including validating the correctness of individual components as well as their integration. Design-by-contract (DBC) offers a promising solution that emphasizes precisely defined and checkable interface specifications for software components. However, existing tools for the DBC paradigm often have some weaknesses: (1) they have difficulty in dealing with dynamically allocated data; (2) specification and checking efforts are disconnected from quality assurance tools; and (3) user feedback is quite poor.

We present Kiasan, a framework that synergistically combines a number of automated reasoning techniques including symbolic execution, model checking, theorem proving, and constraint solving to support design-by-contract reasoning of object-oriented programs written in languages such as Java and C#. Compared to existing approaches to Java contract verification, Kiasan can check much stronger behavioral properties of object-oriented software including properties that make extensive use of heap-allocated data and provide stronger coverage guarantees. In addition, Kiasan naturally generates counter examples illustrating contract violations, visualization of code effects, and JUnit test cases that are driven by code and user-supplied specifications. Coverage/-cost trade-offs are controlled by user-specified bounds on the length of heap-reference chains and number of loop iterations. Kiasan's unit test case generation facilities compare very favorably with similar tools. Finally, in contrast to other approaches based on symbolic execution, Kiasan has a rigorous foundation: we have shown that Kiasan is relatively sound and complete and the test case generation algorithm is sound.

CONTRACT-BASED VERIFICATION AND TEST CASE GENERATION FOR OPEN SYSTEMS

by

XIANGHUA DENG

B.S., Xi'an Jiaotong University, China, 1993
M.S., Kansas State University, 2001

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2007

Approved by:

Co-Major Professor
John Hatcliff

Co-Major Professor
Robby

Abstract

Current practices in software development heavily emphasize the development of reusable and modular software, which allow software components to be developed and maintained independently. While a component-oriented approach offers a number of benefits, it presents several quality assurance challenges including validating the correctness of individual components as well as their integration. Design-by-contract (DBC) offers a promising solution that emphasizes precisely defined and checkable interface specifications for software components. However, existing tools for the DBC paradigm often have some weaknesses: (1) they have difficulty in dealing with dynamically allocated data; (2) specification and checking efforts are disconnected from quality assurance tools; and (3) user feedback is quite poor.

We present Kiasan, a framework that synergistically combines a number of automated reasoning techniques including symbolic execution, model checking, theorem proving, and constraint solving to support design-by-contract reasoning of object-oriented programs written in languages such as Java and C#. Compared to existing approaches to Java contract verification, Kiasan can check much stronger behavioral properties of object-oriented software including properties that make extensive use of heap-allocated data and provide stronger coverage guarantees. In addition, Kiasan naturally generates counter examples illustrating contract violations, visualization of code effects, and JUnit test cases that are driven by code and user-supplied specifications. Coverage/-cost trade-offs are controlled by user-specified bounds on the length of heap-reference chains and number of loop iterations. Kiasan's unit test case generation facilities compare very favorably with similar tools. Finally, in contrast to other approaches based on symbolic execution, Kiasan has a rigorous foundation: we have shown that Kiasan is relatively sound and complete and the test case generation algorithm is sound.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to several people who have been influential in my work. First of all, I wish to thank my major professor, John Hatcliff, for all of his time, helpful guidance, encouragement, and for being infinitely patient with my research. I am also deeply indebted to Robby, who actually has been the main advisor for the research work presented in this thesis for the last three years. His enthusiasm and availability for discussions at any time (including in weekends and sometimes in the middle of the night) are the keys for me to be able to finish my research. I am also thankful to Jooyong Lee, for the wonderful collaborations in the early stage of this research. I would like to thank Matthew Dwyer for advising me for many research topics and I learned a lot from his insightful (and often witty) comments. Finally, I am grateful to Torben Amtoft for many discussions.

Furthermore, I would like to thank Dr. Torben Amtoft, Dr. Matthew Dwyer, Dr. Ruth Miller, and Dr. Mitchell Neilsen along with John Hatcliff and Robby for serving on my committee.

I also like to thank Yu Chen, Adam Childs, Jesse Greenward, Matt Hoosier, Georg Jung, Venkatesh Ranganath, Oksana Tkachuk, Todd Wallentine, and other members of SAnToS group for all the help and nice conversations that are essential to my survival of graduate school.

In terms of sources of encouragement to me, nobody deserves more thanks than my wife, Sue. I am very thankful for her patience and love, for believing in me when I struggled with my research, for being the first guinea pig to read the draft of my thesis. I also would like to thank my daughter, Lois, for filling my life with laugh and joy.

Chapter 1

Introduction

1.1 Motivation

Best practices in software development nowadays heavily emphasize the development of reusable and modular software, which allow software components to be developed and maintained independently. This has improved software development processes employed today and development time and cost are greatly reduced. However, it presents a set of quality assurance challenges. Two of the main challenges are to validate individual component behavior and ensure software compatibility across independently-developed components. In particular, at the time of development and analysis of a component, some components that are needed for the system to run may not be available. We term the systems whose computation structures are incomplete as *open system*. Deep semantical properties of open systems are difficult to reason about because of the missing computation structures.

Object-oriented programming, the most popular programming paradigm today, further complicates the two challenges. Programs written in object-oriented languages, such as Java and C#, usually make extensive use of dynamically allocated objects. In order to analyze object-oriented programs, one has to handle properties that deal with objects, their data, and their relationships. We term such properties as *strong (heap-oriented) properties* as they are hard to analyze due to issues such as aliasing[1]. The dual of strong properties are *lightweight* properties – for example,

[1][51] shows that precise aliasing is undecidable.

simple relationships between scalar values and variable null-ness.

There are many techniques to address the quality assurance challenges. Among them, testing is the most common technique. However, testing requires that test cases are written first. Testing can be used to check but not verify any property (including strong properties) that one is willing to write a test case. However, testing is expensive, it often costs about 50% of the total development cost and time [49]. In addition, it can only be applied when the system is closed, that is, the computation structure is complete. For example, to facilitate testing of open systems, one has to create mock objects [27] to close the systems.

A promising approach to address the challenges of software quality assurance is design-by-contract (DBC). DBC requires enforceable formal interface behavior specifications (contracts) for software components. This allows modular reasoning of individual component behavior because each component can be checked against its contract in isolation. Furthermore, DBC addresses the component integration issue by making sure two communicating components' contracts are compatible. In the context of Java, Java Modeling Language (JML) [41] is the most popular specification language supporting DBC. JML has a compiler that translates JML specifications into Java assertions that are checked during runtime. Essentially, it is a testing technique and has the same weaknesses as testing. JML is also supported by many other tools, for example, ESC/Java [26] which is a static checking tool based on the weakest precondition calculus and theorem proving. It is fully automatic but only checks lightweight properties. Another weakness of ESC/Java is that it uses a pure compositional reasoning methodology that often requires comprehensive specification. In addition, it often depends on bounding loops to terminate the analysis (in the absence of loop invariants). Thus, it is difficult to quantify the behavior coverage that it guarantees. Finally, ESC/Java gives a warning message with a line number when it finds a violation of properties which sometimes is not very intuitive.

To address the quality assurance challenges and overcome shortcomings of existing techniques, we present Kiasan[2], a framework for reasoning about behavioral properties of open (se-

[2]The pronunciation is kē' ah sahn, Indonesian for reasoning with analogy/symbolically.

quential) systems, including strong properties. Our approach is driven by a number of design goals that distinguish it in one or more ways from existing work (*e.g.*, [26, 10, 32, 13]):

G1 *Provides fully automated analysis*: To gain wide-spread adoption from software developers, it is crucial for analysis tools to require no manual intervention.

G2 *Handles strong behavioral properties*: To facilitate reasoning of object-oriented programs, the analysis/tool should be able to precisely check strong properties (*e.g.*, [55]).

G3 *Allows mixed compositional and non-compositional analysis*: While pure compositional reasoning is more scalable, one of its usability problems is that it requires an up-front effort for comprehensive specifications. In the case where an implementation exists or where it is easier to implement than to specify, we should allow one to easily configure the analysis to use the implementation. This allows one to focus on checking the more important parts of the system without undesirable warning or error messages as is the case when using a pure compositional reasoning approach with incomplete/nonexistent specification.

G4 *Has flexibility to adjust analysis cost and coverage*: We believe that an analysis tool should provide enough control over the computational resources that the analysis requires, and it should provide quantifiable behavior coverage guarantees. These allow users to increasingly allocate more resources to gain higher levels of confidence from the tool. For example, when assuring correctness of a method which sorts a list, it is not helpful to use techniques such as iterative deepening in depth-first state-space exploration (*e.g.*, [37, 59]), since the link between a program's behavior coverage and a suitable analysis depth is very difficult to see. Rather, one should be able to specify the behavior coverage, that is, the correctness assurance on lists up to a certain size.

G5 *Provides helpful analysis feedback:* For violations of strong contracts, only only pointing out the program points is not informative. When the analysis finds an error, it should give helpful feedback that explains it, and at the very least, generate an error scenario as evidence.

To meet these design goals, Kiasan combines a collection of insightful designs and engineering decisions that lead to an effective tool that hits a "sweet spot" with respect to the capabilities that software developers would need in practice. More specifically,

- for G1, Kiasan provides a fully automated analysis based on symbolic execution [39] and an extension of symbolic execution for handling objects [37];
- for G2, Kiasan maintains an explicit representation of the visible part of the heap, thus, it can check strong properties;
- for G3, Kiasan uses compositional reasoning when specifications are available, otherwise, it uses implementation directly which reduces the burden of up-front specification;
- for G4, Kiasan uses a longest-reference-chain bound that allows users to adjust the heap configuration coverage and cost;
- for G5, Kiasan has an extension, KUnit, to provide analysis feedback; KUnit automatically generates
    - JUnit tests from contract-annotated method implementations;
    - visualizations of heap objects flowing in and out of methods that can be very useful to developers in understanding complex methods and diagnosing the causes of program errors;
    - branch and bytecode coverage report for generated JUnit tests.

Others [59] have argued that the systematic exploration of heap configurations used by Kiasan and others [37] is computationally intractable for large or complex units. Indeed, previous work on symbolic execution for Java has included very little in the way of experimental results (e.g., only one example is treated in [62]) that would contradict these claims. One of the primary contributions of the work presented in this thesis is to present an experimental study involving twenty-three Java data structure examples which have about 100+ methods (including helper methods) that assesses the performance of not only Kiasan but also two of the algorithms used in the most closely related tools jCUTE [59] and JPF [62] – this is, by far, the most comprehensive study of symbolic execution techniques for object-oriented programs undertaken to this point. Counter to the arguments of [59], our experiments show that for the examples that we considered, KUnit's performance is almost never worse than that of jCUTE [59] and JPF [62], and in most cases it is significantly better (e.g., reducing time to achieve high levels of coverage from multiple hours down to a few minutes).

1.2 Contributions

We list the main contributions of this thesis as follows.

1. *Core Algorithm Improvements:* We have introduced two efficient algorithms for handling the heap data in symbolic execution, *lazier* and *lazier#* initialization algorithms compared to the lazy initialization of JPF [37]. Furthermore, we have a rigorous case analysis of all possible heap shapes generated by several complex data structure operations that establishes the optimality of the *lazier# initialization* algorithm on these data structures under the small *k* bounds.

2. *Rigorous Foundation:* We have formalized the operational semantics of Kiasan and proved that the basic (non-compositional) symbolic executions in Kiasan are relatively sound and complete modulo the bounding strategy and underlying theorem provers. We have also formalized the test input generation algorithms of KUnit and proved that they are relatively sound (guaranteed to generate tests for all execution paths up to the bounding strategy).

3. *Implementation of Kiasan and KUnit:* We have implemented Kiasan on top of the Bogor model checking framework [54, 3]. Kiasan can check strong heap properties of open systems with or without contracts. Kiasan uses a longest-reference-chain bounding strategy which provides better heap configuration control compared to other approaches that bound on the numbers of search steps or numbers of symbolic objects in the heap. We have also

implemented an analysis feedback plugin, KUnit, for Kiasan. KUnit can generate JUnit test cases (including mock objects for open systems), visualization of input/output heap graphs, and branch/statement coverage metrics.

4. *Extensive Experimental Study:* We have done an empirical evaluation (using twenty-three different data structure packages) of JPF's *lazy initialization* algorithm, Kiasan's *lazier initialization* algorithm, and *lazier# initialization* algorithm. It shows that the lazier# initialization algorithm significantly improves upon the lazier initialization algorithm, which in turn significantly improves upon JPF's lazy initialization algorithm. We found that KUnit is very efficient with heap data: it can obtain 100% feasible branch coverage on almost all of our 23 heap intensive examples. In addition, we have compared the performance of jCUTE [59] and KUnit on the same set of examples. We found that KUnit and jCUTE have comparable performance on simple heap manipulation examples; but KUnit performs significantly better than jCUTE on complex heap manipulation examples such as the red-black tree and AVL tree.

1.3 Acknowledgments

Involvement with Kiasan My involvement with Kiasan was accidental. In summer 2005, Jooyong Lee, a Ph.D student from BRICS (Denmark), came to our group as an exchange student. Robby advised him to work on checking strong properties of open systems based on symbolic execution. Just after they started, they needed to build an interface with theorem provers because symbolic execution uses theorem provers to decide whether a path is infeasible (symbolic execution is presented in Chapters 2 and 3). At that time, I was stuck on my PhD research topic: static bug finding based on patterns. Since I have some experiences on interfacing with theorem provers, I volunteered to help them building the interface. It is such an interesting research topic, and Jooyong and Robby were extremely nice to work with. I became more involved with the Kiasan project. Finally, I focused my research solely on Kiasan.

Initial Verification Effort of Kiasan Kiasan builds upon the JPF's lazy initialization algorithm [37]. However, Kiasan uses a different bounding strategy, the longest-reference-chain bound, and fixes unsoundness issues in [37]. Kiasan also has an improved core algorithm: lazier initialization algorithm. Furthermore, Kiasan extends the single method checking into compositional checking and identifies two important techniques that enable the compositional checking. Jooyong, Robby, and I worked very closely on all these ideas. In fact, we had at least one meeting a day. Overall, I did about 70% of the implementation and almost all the examples. Later, we worked together to setup a formalization framework for Kiasan and I solely proved the relative soundness and completeness of Kiasan with guidance from Robby. The result of this effort is published as [23].

Improvements of Kiasan After Jooyong's departure, I became the sole maintainer of Kiasan code base. At that stage, Kiasan's user feedback was quite poor: it only gave a line number when an error was found which is similar to what ESC/Java does. This made adding examples to Kiasan quite challenging. John Hatcliff suggested to conduct larger case study to further assess Kiasan. To make debugging examples a little easier, we extended Kiasan's algorithm to generate helpful user feedback including input/output visualization, JUnit test cases, and statement/branch coverage report. I proposed a novel "modified backtracking rule" approach to generate effective pre-states. I also formalized the backtracking rules and input generation algorithm; and proved its soundness. In addition, I did all the implementation for this work. With the help of KUnit, I was able to accomplish an extensive experimental study which shows that Kiasan is a significant improvement over the original JPF lazy initialization. The result of this effort is published as [24].

When I debugged complex data structures such as red-black tree, I found it was very hard to verify the correctness of Kiasan's output (for $k = 3$, there are hundreds of cases). Initially, I manually inspected all the cases (up to $k = 2$) which was very labor intensive and did not even scale to $k = 3$. I felt that we need a theoretical way to at least quantify the number of cases. Another related question was that we knew lazier initialization improves dramatically over lazy initialization; but can lazier initialization be even better? I developed a method to count the theoretical numbers of non-isomorphic states for several complex data structures such as binary search tree, AVL tree, red-black tree, etc. This method uses a standard combinatorics technique, *generating functions* [64], to simplify complex recurrence relations or even further–get closed forms for those relations. The result of this method shows that the lazier algorithm is sub-optimal for binary search tree and red-black tree, which means that it generates more cases than that are computed by the method. I investigated the examples and identified the root of the problem. Robby, John Hatcliff, and I came up with an even lazier algorithm and we named it lazier# initialization algorithm which was validated under small k by an extensive experimental study. I also formalized the semantics of lazier# algorithm and proved its relative soundness and completeness. The result of this effort is published as [25].

1.4 Organization

The rest of this thesis is organized as following.

- Chapter 2 presents background information about symbolic execution, more specifically, the original symbolic execution technique proposed by King [39] and a recent extension of symbolic execution to objects in object-oriented languages, *lazy initialization* algorithm.

- Chapter 3 discusses all the aspects of symbolic execution in Kiasan. We present two improved core algorithms (lazier and lazier# initialization algorithms), and compositional checking technique in Kiasan. Finally, we show the implementation of Kiasan.

- Chapter 4 contains the combinatorics analysis for optimal numbers of cases for several complex data structures and algorithms using a standard combinatorics technique, *generating functions*. We present the analysis for binary search tree, AVL tree, red-black tree, etc.

- Chapter 5 presents the formalization of Kiasan. The formalization includes the operational semantics for symbolic executions with lazy/lazier/lazier# initialization algorithms and concrete execution in Java Virtual Machine (JVM). Furthermore, the soundness (for each symbolic trace, there exists a corresponding concrete trace) and completeness (for each concrete trace, there exists a corresponding symbolic trace) are proved.

- Chapter 6 discusses the analysis feedback plugin, KUnit, for Kiasan. KUnit uses modified backtracking algorithms for constructing effective symbolic input states and then concretizes the effective symbolic input states to gets test inputs. Then we show the generation of additional artifacts such as JUnit test cases, input/output graphs, and mock objects for open systems.

- Chapter 7 presents the formalization of KUnit input generation algorithms and shows the soundness proof of generated JUnit test cases.

- Chapter 8 presents three examples to demonstrate the specification in Kiasan and generated output. The examples are insertion sort (array based), binary search tree, and red-black tree.

- Chapter 9 shows an extensive experimental study and comparisons with JPF and jCUTE. It also includes a `java.util.TreeMap` coverage report.

- Chapter 10 presents the related work and Chapter 11 concludes.

Chapter 2

Background

In this chapter, we describe the basic (scalar) symbolic execution techinque and a recent developement of symbolic execution for handling objects in object-oriented languages such as Java: *lazy initialization* algorithm.

2.1 Symbolic Execution

In a 1976 paper[39], King proposed symbolic execution which is a technique for executing code modules that treats input parameters and globals as symbolic values. Since symbolic values are introduced, there are two changes to program analysis. First, the domain of values has to be augmented to include symbolic values. Second, besides a mapping from variables to values, each state of symbolic execution contains a boolean predicate (ϕ) called *path condition* to record the condition that directs the execution to follow the current path.

Symbolic execution systematically explores all feasible paths of the program. The result of the exploration forms an computation tree. We will use the absolute value (`abs`) method shown in Figure 2.1 as an example to illustrate the basic ideas of symbolic execution.

```
int abs(int x) {
  if(x < 0)
     x = -x;
  if(x < 0)
     assert false;
  return x;
}
```

Figure 2.1: *Absolute Value Method*

The symbolic execution of the `abs` method will generate a symbolic computation tree as shown in Fig-

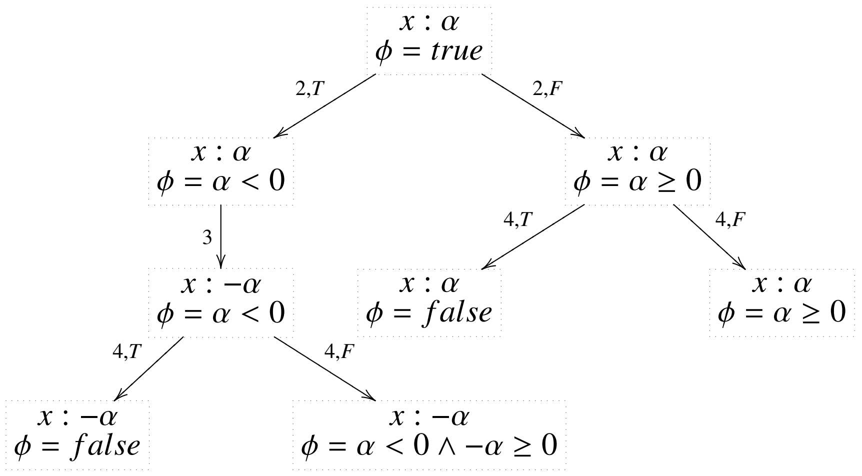


Figure 2.2: *Absolute Value Method Symbolic Execution Tree*

ure 2.2. In the tree, each node is a state. For example, the initial state (the root node) for the `abs` method is that x has a symbolic value α and the path condition is `true`. When executing line 2, the symbolic execution does not have sufficient information to decide which branch to take because $x < 0$ and $\neg(x < 0)$ are satisfiable given current path condition `true`. So it will take both branches and add branching conditions to the path conditions as indicated in the tree where the root node has two children with edges $2, T$ and $2, F$ which lead to `true` branch or `false` branch respectively. If symbolic execution takes the `true` branch, the path condition becomes $\alpha < 0$. Then after executing statement 3, the state becomes $x = -\alpha$ and the path condition remains the same $\alpha < 0$. At statement 4, a branching occurs again. But after taking the `true` branch, symbolic execution finds out that the path condition becomes $\alpha < 0 \wedge (-\alpha) < 0$ which is `false`. The path is infeasible, hence abandoned. So only the `false` branch of statement 4 is taken. Execution follows the `false` branch of statement 2 is similar. In all cases, the `true` branch of line 4 is always infeasible, and thus statement 5 is never reached. King also described a commutative propertycommutative property of symbolic execution (without proof): the operation of replacing symbols with concrete integers and the operation of executing program are commutative. Figure 2.3 illustrates the commutativity where $P(X)$ is a program under analysis with X an input parameter; K is a concrete value (in this case, an integer); $E(P(X))$ is the result of symbolically executing P; and $E(P(K))$ is the result of executing P with input K. This commutative property actually is a part of the simulation relation between concrete and symbolic executions presented in Chapter 5.

Generalized Symbolic Execution King [39] proposed symbolic execution with integer types as inputs. But as object-oriented languages such as Java become popular, it becomes imperative to extend the symbolic execution technique to handle objects. Recently, Khurshid etc.[37] proposed a generalized symbolic execution algorithm called *lazy initialization* which deals with object inputs. Lazy initialization treats input objects as symbolic objects with each field uninitialized until the field is accessed. The initialization algorithm for a field f of a symbolic object o is listed as following:

$$\begin{tikzcd}
P(X) \arrow[r, "X \leftarrow K"] \arrow[d, Rightarrow, "\text{Symbolic execution}"] & P(K) \arrow[d, Rightarrow, "\text{Concrete execution}"] \\
E(P(X)) \arrow[r, "X \leftarrow K"] & E(P(K))
\end{tikzcd}$$

Figure 2.3: *Commutativity Diagram*

```
if type of o.f, T, is primitive
   o.f:= a new symbolic primitive value
else
   o.f is nondeterministically assigned to null,
        a new symbolic object of type T, or one of the existing heap obje
        that have the type T
```

Chapter 3

Symbolic Executions in Kiasan

In this chapter, we discuss symbolic executions in Kiasan and implementation of Kiasan. Section 3.1 explains the symbolic execution with lazy initialization in Kiasan and two consecutive core algorithm improvements over the lazy initialization: *lazier* and *lazier#* initialization. To guarantee the termination of Kiasan, we use a longest-reference-chain bounding technique presented in Section 3.2. Section 3.3 presents the compositional checking in Kiasan. Finally, Section 3.4 shows the implementation of Kiasan framework.

Acknowledgments

This chapter is based on two papers: first titled "Bogor/Kiasan: A *k*-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems" by Xianghua Deng, Jooyong Lee, and Robby that appeared in the Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering [23]; second titled "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs" by Xianghua Deng, Robby, and John Hatcliff to appear in the Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods [25].

```
public class Node<E> {
  //@ ensures data == \old(n.data) && n.data == \old(data);
  public void swap(@NonNull Node<E> n)
  { E e = data; data = n.data; n.data = e; }
  private Node<E> next; E data;
}
```

Figure 3.1: *A Swap Example*

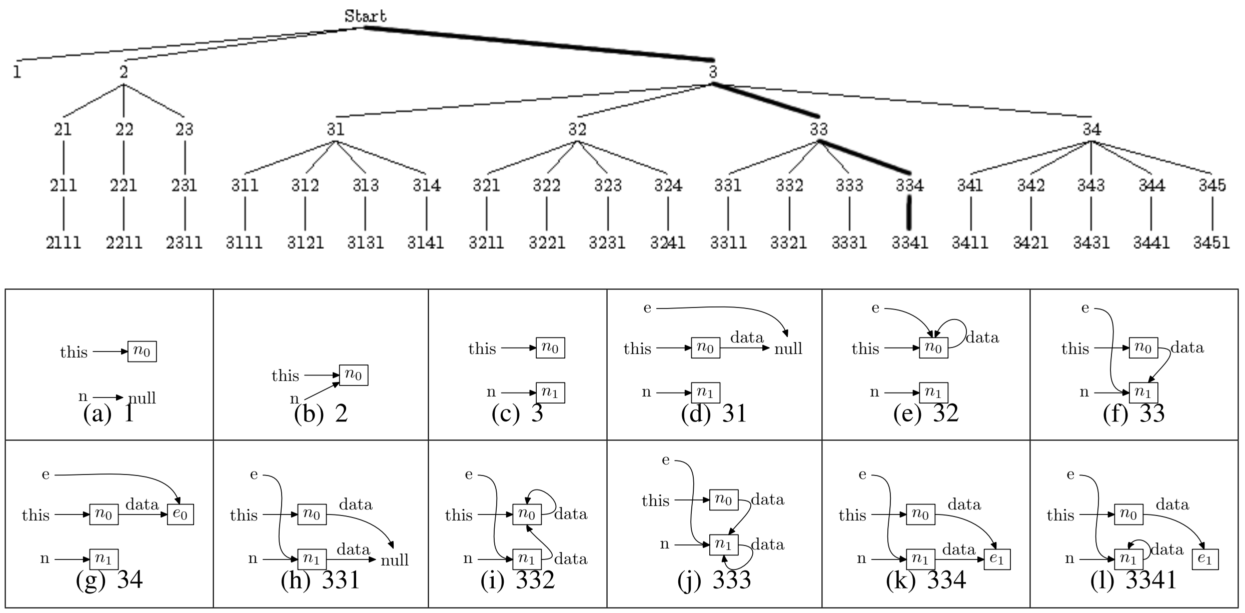


Figure 3.2: *Lazy Symbolic Execution Tree and An Example Trace (3-33-334-3341 and Sibling States)*

3.1 Basic Symbolic Execution

In this section, we describe our basic (non-compositional) and stateless symbolic execution technique. Essentially, given a method, we start with a symbolic state where all method parameters and global variables are assigned symbolic values (or non-deterministically NULL and all possible aliasings for reference variables). For primitive values, we keep track of their relations to other (symbolic and/or concrete) values during the method execution, *i.e.*, path conditions. For objects and arrays, we take the *lazy initialization* algorithm described in Section 3.1.1 as the starting point and develop two successively improved algorithms: *lazier initialization* and *lazier# initialization* discussed in Sections 3.1.2 and 3.1.3.

3.1.1 Lazy Initialization

To handle unknown heap structures, Kiasan uses an enhancement of the *lazy initialization* algorithm originally introduced in [37]. The lazy initialization algorithm starts with no or partial knowledge of object values (i.e., *symbolic objects* whose fields are uninitialized) referenced by program variables. As the program executes and accesses object fields, it "discovers" (i.e., *materializes*) the field values on an on-demand basis (i.e., hence the term "lazy initialization"). When an unmaterialized field is read, if the field's type is a primitive, then a fresh symbol is created for that scalar value. Otherwise, for an unmaterialized reference field, the algorithm systematically (safely) explores all possible points-to relationships by non-deterministically choosing among the following values for the reference: (a) NULL, (b) any existing symbolic object [1] whose type is compatible with the field's type, or (c) a fresh symbolic object (whose type is constrained to be equal to or a subtype of the field's type).

To illustrate lazy initialization, consider the following `swap` method for `Node` in Figure 3.1. The top part of Figure 3.2 illustrates the symbolic execution computation tree built using lazy initialization. To save space in the display of the tree, we represent each tree node (system state) by a unique label corresponding to the path through the tree to the current code. The bottom part of Figure 3.2 shows heap configurations for some of the states in the computation tree.

To generate the computation tree of Figure 3.2, the symbolic execution begins with a non-deterministic choice of possible aliasing between the method parameter `n` and the `this` reference (i.e., States 1, 2, and 3). Note that both the `next` and the `data` fields of `this` and `n` are unknown (unmaterialized). Out of the three cases, State 1 does not satisfy the `@NonNull` precondition for `n`, thus it is not considered further. Now, consider the sub-tree starting from State 3. Upon executing `swap`'s first statement, the `this.data` field is now materialized according to the lazy initialization algorithm described above; it non-deterministically chooses the value of `this.data` to be: NULL (31), equal to `this` – n_0 (32), n_1 (33), or a fresh symbolic object e_0 (34). Let us continue

[1]The algorithm does not choose from concrete objectswhich is created in the method because the object that the field really "points to" should exist before the method entry, but concrete objects are created after the method entry.

with the sub-tree starting from State 33. Upon executing `swap`'s second statement, the algorithm non-deterministically chooses the NULL value, n_0, n_1, or a fresh symbolic object e_1 for the `n`'s `data` field, thus, resulting in the States 331, 332, 333, and 334, respectively. Executing `swap`'s last statement from 334 produces 3341 (the trace 3-33-334-3341 is highlighted in Figure 3.2). Note that the symbolic computation tree characterizes all possible concrete executions of `swap`; Kiasan's lazy initialization algorithm has been formalized and its relative soundness and completeness has been proven in Chapter 5. In addition, all `swap`'s post-states in Figure 3.2 satisfy `swap`'s postcondition, thus, we conclude that the postcondition always holds (checking the postcondition requires reconstructing the effective pre-state of each post-state 6).

Handling Arrays: Arrays present a unique challenge: the length of an array may be unknown. In addition, arrays can be accessed by a symbolic (unknown) integer index. To address these issues, we model an array as an accumulator that remembers the set of indexes that have been accessed and their corresponding values. Initially, the accumulator is empty. If the array is accessed with an index v (concrete or symbolic integer), v is compared with the already accumulated indices: if v is equal to one of them, then we use its corresponding value; otherwise a new entry with a fresh symbolic value X is stored with v as its index and X is returned. Similarly to field lazy initialization, this cuts down the number of paths that have to be explored, *i.e.*, instead of comparing the index being accessed with all the indexes of the array (which can only be tractably done if the length of the array is bounded), we only compare the index with what have been accessed.

3.1.2 Lazier Initialization

As we can observe, the lazy initialization algorithm produces a rather large state-space even for `swap`. In [23], we introduced an optimized algorithm called *lazier initialization* based on the observation that when an uninitialized reference type variable is first read, it is not necessary to resolve the aliasing/object value at that particular point; only when the object referenced by the variable is accessed, that is when it is necessary to resolve the value. Basically, the lazier algorithm divides the lazy initialization into two steps as follows. Step 1, when an uninitialized reference

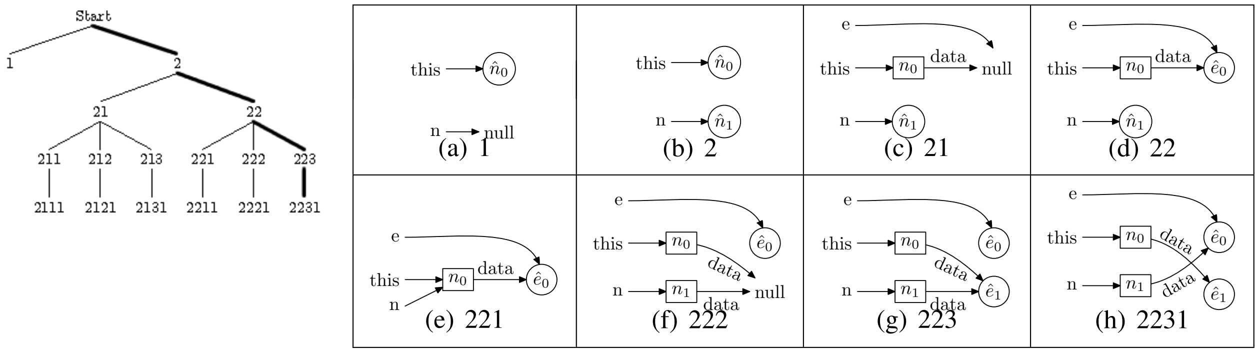


Figure 3.3: *Lazier Symbolic Execution Tree and An Example Trace (2-22-223-2231 and Sibling States)*

type variable is read, it is lazier-ly initialized with the NULL value or a fresh *location* (whose type is the same as the variable's type); in essence, the symbolic location represents all possible objects that may be referenced by the variable (i.e., it *abstracts* such a set of objects). Non-reference-type variables are handled similarly to the lazy algorithm. Step 2, when a field of a symbolic location is accessed (read), (a) the symbolic location is then replaced by non-deterministically choosing any existing object or a fresh symbolic object (with compatible type); if the access is a read access and the field is unmaterialized, (b) the field is then initialized (with the NULL value or a fresh symbolic location). The effects of these two steps are: (1) delaying the non-deterministic choice of objects in the lazy algorithm, and (2) the second step may not be needed in some cases. Thus, it produces a (significantly) smaller state-space (see our experiment data in Chapter 9).

To illustrate the lazier algorithm, let us reconsider the `swap` example in Figure 3.1. The left hand side of Figure 3.3 illustrates the symbolic computation tree using lazier initialization; the highlighted path in the (lazier) computation tree corresponds to the highlighted path in the (lazy) computation tree shown in Figure 3.2 (i.e., it simulates the lazy path). Symbolic locations are annotated with $\hat{\cdot}$. Similar to the lazy algorithm, the lazier initialization algorithm starts with a non-deterministic choice. However, there are only two choices instead of three in the beginning. State 1 in Figure 3.2 is abstracted into State 1 in Figure 3.3, and State 2 and 3 in Figure 3.2 are abstracted into State 2 in Figure 3.3 (i.e., both $\hat{n}_0$ and $\hat{n}_1$ may actually be n_0 or n_1). When $\hat{n}_0$'s data field is read at `swap`'s first statement, $\hat{n}_0$ is replaced with n_0 (there is no existing symbolic object for the non-deterministic choice, thus, it uses a fresh symbolic object), and n_0's `data` field is initialized with either the NULL value (21) or a fresh symbolic location $\hat{e}_0$ (22). From State 22, there are three possible choices when executing `swap`'s second statement. We first replace $\hat{n}_1$ by non-deterministically choosing the existing object n_0 or a fresh symbolic object n_1. In the former case, the `data` field has been initialized, thus no special treatment is needed (221). In the latter case, n_1's data field is "lazier-ly" initialized with either NULL (222) or a fresh symbolic location e_1 (223). Executing `swap`'s last statement from 223 produces 2231. Note that 2231 in Figure 3.3 safely approximates 3341 in Figure 3.2.

As we can observe, the computation tree in Figure 3.3 which has 16 states and 6 paths, is much smaller than the one in Figure 3.2 which has 50 states and 20 paths, because the non-deterministic choices for `this`, `n`, and the `data` fields are delayed, and the second steps of the lazier initialization for `data` field accesses never happen. Moreover, all `swap`'s post-states in Figure 3.3 still satisfy `swap`'s postcondition, thus, we conclude that the postcondition always holds. Note that we do not need to replace $\hat{e}_0$ and $\hat{e}_1$ with symbolic objects when checking the postcondition, as they will be compared against themselves (i.e., `data==\old(n.data)` iff $\hat{e}_1 = \hat{e}_1$). Kiasan's lazier initialization algorithm has been formalized and proved that it simulates the lazy initialization algorithm as shown in Chapter 5 (i.e., it is relatively sound and complete).

3.1.3 Lazier# Initialization

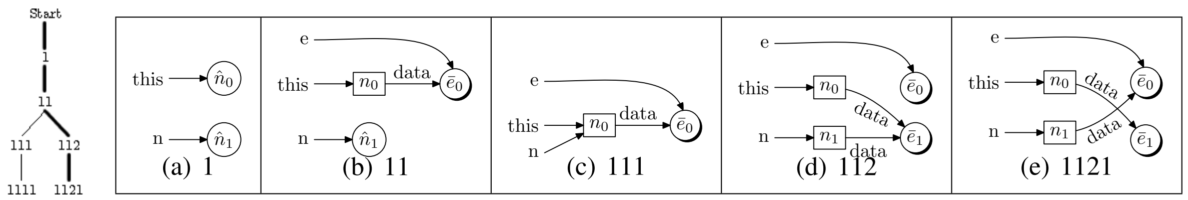


Figure 3.4: *Lazier# Symbolic Execution Tree and An Example Trace (1-11-112-1121 and Sibling States)*

As described previously, the lazier initialization algorithm significantly reduces the state-space for symbolic execution of object-oriented programs while still preserving strong heap-oriented properties. However, one might wonder whether it can still be improved. More specifically, we are interested in investigating whether the algorithm is *case-optimal* – it considers the minimum number of behavior cases (i.e., pairs of pre/post-states) when analyzing a given property and example (e.g., it considers only non-isomorphic heap shapes). Clearly, the answer is problem-dependent (and size-dependent for programs working with possibly unbounded number of objects). We have leveraged a combinatorics technique, *generating functions*, to calculate the minimum numbers of cases for different k-bounds for the binary search tree, AVL tree, and red-black tree. Detail calculation is shown in Chapter 4. For example, for $k = 1$, lazier initialization generates 12 cases for the `insert` method of the binary search tree example; but the calculated number is 4. We conclude that there is an inefficiency in the lazier initialization algorithm.

```
 1 BinaryNode<T> insert(T x, BinaryNode<T> t) {
 2   if (t == null)
 3     t = new BinaryNode<T>(x, null, null);
 4   else if (comparator.compare(x, t.element) < 0)
 5     t.left = insert(x, t.left);
 6   else if (comparator.compare(x, t.element) > 0)
 7     t.right = insert(x, t.right);
 8   else
 9     ; // Duplicate; do nothing
10   return t;
11 }
```

Figure 3.5: *A Binary Search Tree Insertion*

To address the inefficiency of the lazier initialization algorithm, we have developed an even lazier initialization algorithm which we named the *lazier# initialization* algorithm. We observed that one source of efficiency in the lazier initialization algorithm is due to the fact that it is optimized for non-NULL variables; it optimistically assumes most variables are non-NULL (this is in-line with JML's default invariant for reference type variables). That is, it eagerly initializes an uninitialized (reference type) variable as NULL or a fresh symbolic location upon access. For example, consider the source code of `insert` shown in Figure 3.5, the lazier initialization algorithm non-deterministically chooses between NULL and a fresh symbolic location for the field `t.element` at line 4. However, the `t.element` is only used when comparing with the inserted element by `comparator.compare`, and the Java `Comparator` interface does not require `compare`'s parameters to be non-NULL. Thus, whether the value is NULL or non-NULL is irrelevant (i.e., processing the `compare` interface following a compositional checking approach to check the `insert` implementation will produce either negative, zero, or positive regardless). Therefore, the non-deterministic choice is too early at line 4 in the sense that it unnecessarily exposes details about the heap objects.

In the lazier# initialization algorithm, we introduce an intermediate step by initializing such variables with a new flavor of symbolic object that abstracts NULL as well as any object of the appropriate type. We use the general term "symbolic references" (with annotation $\bar{\cdot}$) for abstract values that abstract both NULL and any object of the appropriate type, and we use the term "symbolic locations" (with annotation $\hat{\cdot}$ as used previously) to refer to non-NULL values (i.e., we now have three abstraction levels for objects: (1) symbolic objects, (2) symbolic locations, and (3) symbolic references). Thus, the lazier# algorithm can be described as follows. Step 1, when an unmaterialized variable is read, it is initialized with a fresh symbolic reference (i.e., there is no non-deterministic choice). Step 2, when a field of a symbolic reference is accessed, the symbolic reference is replaced with NULL (which results in raising a null dereference exception), or a fresh symbolic location. In the case of the latter, the algorithm proceeds similarly to the lazier initialization algorithm (but, an uninitialized field is lazier#-ly initialized instead lazierly initialized). Note that the first step can be further optimized (*NV*:) by directly using a fresh symbolic location if the variable is known to be non-NULL.

To illustrate the lazier# initialization algorithm, let us revisit the `swap` example. Figure 3.4 illustrates the symbolic computation tree using the lazier# algorithm and a trace (along with its states and their sibling states) that simulates the highlighted trace in Figure 3.3 (and thus, it simulates the trace highlighted in Figure 3.2). The algorithm starts with one state (State 1). Notice that `n` refers to symbolic location $\hat{n}_1$ instead of a symbolic reference because of the *NV* optimization mentioned above (without the optimization, we use a fresh $\bar{n}_1$). When executing `swap`'s first statement, $\hat{n}_0$ is replaced with a fresh symbolic object n_0, and its `data` field is initialized with a fresh $\bar{e}_0$, thus resulting in State 11. Continue on with executing `swap`'s second statement, $\hat{n}_1$ is replaced with either the existing symbolic object n_0 or a fresh symbolic object n_1. In the former case, n_0's `data` field has been initialized, thus no special treatment is needed (111). In the latter case, n_1's `data` field is initialized with a fresh symbolic reference $\bar{e}_1$ (112). From 112, it produces

1121. As we can observe, the lazier# computation tree in Figure 3.4 realizes a correct abstraction of both the lazy (Figure 3.2) and lazier (Figure 3.3) computation trees while still exposing enough information to establish `swap`'s postcondition.

3.2 *k*-bounding

There are two main challenges when using symbolic execution: (1) the termination of and (2) the scalability of the algorithm. To address these issues, Kiasan [23] incorporates a different bounding technique to help manage symbolic execution's complexity, while providing fine-grained control over parts of the heap that one is interested in. In essence, we bound the sequence of lazy/lazier/lazier# initializations originating from each initial symbolic object (longest-reference-chain) up to a user-supplied value k. This user-adjustable bounding provides an effective and controllable trade-off between analysis cost and behavioral coverage. When using a bound k, the analysis can guarantee the correctness of a program on any heap object configuration with reference chains whose lengths are at most k. In the case where the analysis does not exhaust k, a complete behavior coverage is guaranteed. To handle diverging loops (or recursions), we limit the number of loop iterations *that do not (lazily) initialize any heap object*. That is, we prefer exhausting the k-bound first to try to achieve the advertised heap configuration coverage.

3.3 Contract-based Symbolic Execution

To reason about open systems, we employ contract-based reasoning often used in compositional analysis techniques such as ESC/Java [26]. When we analyze a method M, we require that M's contract is transformable to an executable form similar to [16]. Intuitively, when analyzing M, Kiasan assumes M's effective `pre` at the method entry and asserts M's effective `post` at method exits. To achieve this using symbolic execution, Kiasan creates a wrapper method for M that:

1. `assumes` M's executable `pre`,

2. calls M and stores M's return value (if any) to a temporary variable x,

3. `asserts` *M*'s executable `post` (that uses *x*'s value in place of the return value.)

In essence, executing (1) sets up the symbolic states according to *M*'s `pre` (*e.g.*, they initialize the heap appropriately), and states non-conforming to (1) will be ignored. Executing (3) checks whether the resulting states from (2) satisfy *M*'s `post` (if `post` cannot be ensured from the path condition, then an error is raised). Kiasan can check `post` referring to pre-state's values (*i.e.*, values at method entry), and JML's `modifies`, `assignable`, and `\fresh` similar to [55].

Instead of directly executing method calls from *M*, Kiasan uses contracts in place of the actual implementations of open-ended methods (user-configurable). Intuitively, if *M* calls an open-ended method *N*, it checks whether *N*'s `pre` is satisfied; an error is raised if that is not the case. If it is satisfied (or if none is specified), Kiasan uses *N*'s `post` to determine the effects of the method call. To do this, for each open-ended method *N* called by *M*, it creates a stub for *N*, and redirects the corresponding method call to *N* to call the stub instead; the stub consists of a sequence of statements that:

1. `asserts` *N*'s `pre`,

2. removes values from modified fields stated in *N*'s contract (hence their values become undefined), if the modified fields do not refer to fresh objects created in *N* (as specified in *N*'s contract using a similar construct such as JML's `\fresh`); otherwise, fresh symbolic objects are created for such fields,

3. pushes a symbolic reference in *M*'s stack for non-primitive return type and a symbolic value for primitive return type,

4. `assumes` *N*'s `post`.

In essence, executing (2) drops information about fields that are modified by *N*, and executing (4) initializes them with values which satisfy *N*'s `post`.

We can summarize method calls similarly to [32]; that is, we cache method results and their corresponding contexts; if subsequent calls use the same context, we use the cached results.

To close the environment, Kiasan creates a driver for M that starts with a symbolic state where all method parameters and fields that are possibly referenced by M are initialized with primitive symbolic values or symbolic references, according to their types. The analysis proceeds with symbolically executing the driver for M. The subsequent subsections describe two techniques that improve Kiasan's performance as well as enable us to check strong heap properties in the context of analyzing open systems.

3.3.1 Heap Region Versioning

```
public class LinkedList<E> {
  //@ inv: isAcyclic();
  @NonNull LinkedNode head = new LinkedNode();
  /*@ pre: isSorted(c)
    @      && other.isSorted(c);
    @ post: isSorted(c);
    @*/
  void merge(@NonNull LinkedList<E> other,
             @NonNull Comparator<E> c) {
    LinkedList<E> ll = new LinkedList<E>();
    LinkedNode n1 = this.head.next;
    LinkedNode n2 = other.head.next;
    while (n1 != null && n2 != null) {
      if (c.compare(n1.data, n2.data) < 0) {
        ll.addLast(n1.data); n1 = n1.next;
      } else {
        ll.addLast(n2.data); n2 = n2.next;
      }
    }
    while (n1 != null) {
      ll.addLast(n1.data);
      n1 = n1.next;
    }
    while (n2 != null) {
      ll.addLast(n2.data);
      n2 = n2.next;
    }
    head = ll.head;
  }
  class LinkedNode {
    E data;  LinkedNode next;
  } }
```

Figure 3.6: *A Merge Example (excerpts)*

Figure 3.6 [2] presents a sorted list merge example that motivates our approach. Intuitively, the `merge` method's contract indicates that given a non-null and sorted (from the preconditions `@NonNull` and `pre`) acyclic list (from the invariant `inv`) with respect to the specified `Comparator c`, the method merges the contents of that list into the receiver object (given that it is also sorted) and as the result, the receiver object is also a sorted acyclic list (`isAcyclic` and `isSorted` are pure, i.e., they do not modify existing objects). We highlight one challenge when reasoning about such programs and specifications.

The `compare` method is *open-ended*, *i.e.*, in contrast to reasoning about a complete system such as [37] where we know the actual objects and the data being manipulated, we do not know the actual implementation of the method (or even if there is an implementation for the type that will substitute `E`). This is also in contrast with some systems [42, 65, 38, 6], where elements being compared

[2] The formats of annotations (pre/postconditions and invariants) are for illustration purposes. Kiasan uses slightly different annotations.

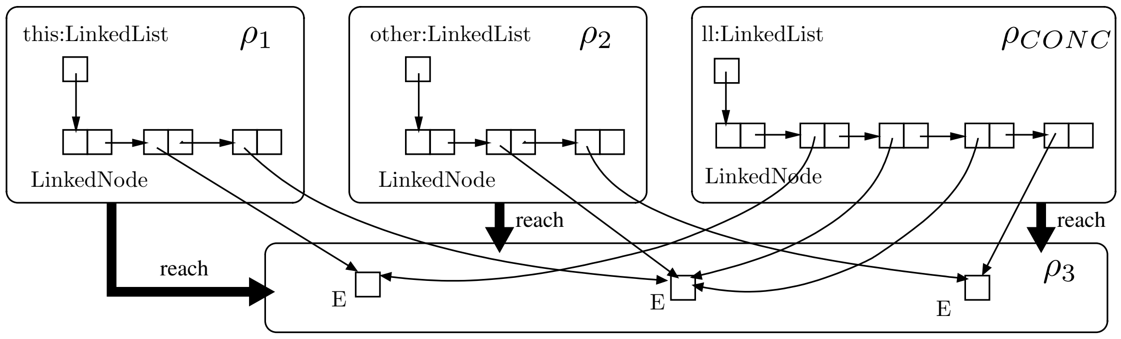


Figure 3.7: *A Region Relation Scenario of* `merge`

are of scalar types or immutable objects. Since we can not decide which data will be used for comparison, we have to be conservative and assume that it will use all the data that the method can reach.

We have to establish that whatever information used for comparisons must not be modified by `merge` (or methods called from it) to ensure the comparisons done later in postcondition are unaffected. (This is a bit too strong as it is fine to modify any information that will be accessed by `compare` if it does not change `compare`'s result; we only consider the former case in this thesis.) Otherwise, there is no guarantee that the receiver object is sorted afterward. For example, suppose that we insert a code just before the end of `merge`. We need to check whether the inserted code invalidates the elements' ordering. This can be detected by using heap region separation. That is, the inserted code cannot invalidate the ordering if it does not modify the element objects. However, establishing this requires a precise heap analysis that is able to leverage, for example, heap region information.

Consider the scenario depicted in Figure 3.7 where each list object is in its respective region ρ_1 and ρ_2, `this`$\neq$`other`, and the list elements are in a separate region ρ_3. It indicates that objects in ρ_3 cannot reach objects in ρ_1 and ρ_2.

Enhanced with this region specification, the analysis starts with two fresh symbolic locations pointed by `this` and `other`, tagged with ρ_1 and ρ_2 as their region descriptors, respectively. Freshly created concrete objects are tagged with a special region ρ_{CONC}. When a field f_ρ is materialized, it can point to objects from region ρ, the choosing range of the existing symbolic objects will be existing symbolic objects in region ρ. This reduces the number of aliasing cases, thus improves the performance of the analysis. Furthermore, we associate a version number to each region that is incremented when any object in the region is updated. This allows us to detect that subsequent method calls such as `compare` whose context is in the same region and version, return the same result values. That is, we cache the method calls, and return previously computed result values if there is no change in any of the objects inside the regions reachable from the context. Therefore, we are able to conclude that `merge`'s `post` holds, because `compare` is pure, and `merge` and `addLast` do not update ρ_3's version.

Modeling of Comparator and Comparable

According to Java 1.5 API, `Comparable.compareTo` and `Comparator.compare` are pure and return either a negative integer, zero, or a positive integer; furthermore they impose a total order on a set of objects. This is crucial because object ordering relationship is often used in programs manipulating data structures. The transitivity property of the total order cannot be specified in an executable form because establishing the transitivity property requires the execution history. To facilitate this, we provide a specification pattern for describing transitive closure relation on objects (*e.g.*, via method calls). First, we define a function

$$\text{comparison-Id(o)=object-Id(o)+region-Id(o)+ region-Version(region-Id(o))},$$

which captures the object identity together with the region identity and version. Then we let

$$\alpha = \texttt{Comparator.compare(o1,o2) or o1.comparesTo(o2)},$$

where α is a fresh integer symbol, and add the following formula into the path condition:

$$\begin{aligned}\alpha < 0 &\iff \text{comparison-Id(o1)} < \text{comparison-Id(o2)}\wedge\\ \alpha = 0 &\iff \text{comparison-Id(o1)} = \text{comparison-Id(o2)}\wedge\\ \alpha > 0 &\iff \text{comparison-Id(o1)} > \text{comparison-Id(o2)}.\end{aligned}$$

Empowered with this, Bogor/Kiasan successfully checks the strong properties of `merge` shown in Figure 3.7 and other examples shown in Chapter 9.

3.3.2 Context Versioning

```
1   public class W {
2     MyInt myInt;
3     void foo(@NonNull W wrapper) {
4       MyInt m = new MyInt();
5       int tmp = wrapper.myInt.f;
6       W w = bar(m);
7       if(w.myInt != m){...}
8     }
9     //@ post: \result.myInt != null;
10    @Fresh W bar(@NonNull MyInt m) {
11      W w = new W(); w.myInt = m; return w;
12    }
13  }
14  class MyInt { int f = 0; }
```

Figure 3.8: *A Context Versioning Example*

Recall that in the original lazy initialization algorithm, when it chooses existing objects, it only chooses from heap objects that are symbolic. This produces unsoundness in the context of contract-based reasoning described earlier. The example in Figure 3.8 demonstrates this problem. Suppose we are analyzing the method `foo` where we use `bar`'s specification instead of its implementation. Since `wrapper` is a non-null parameter, the analysis starts with a symbolic location for it. The field access `wrapper.myInt.f` at line 5 will make `wrapper.myInt` initialized and the choosing range does not include the local `MyInt` object created at line 4. This is consistent because `wrapper.myInt` at line 5 can only point to any object from the calling context of `foo`. For the method invocation at line 6, contract-based reasoning creates a symbolic object (because of `@Fresh`) for the return value of `bar` and assigns it to `w`. After the method call at line 6, `w.myInt` points to the object *o* created at line 4 in real execution. However, according to the lazy initialization algorithm, the choosing range unsoundly excludes *o* because it is a concrete object.

To address this, instead of using one bit flag attached to an object to indicate whether it is concrete or symbolic in implementation, we record context version by using an integer value attached to each object. Each time a method invocation uses the method's specification in place of its implementation, the objects created after the invocation have a higher version number than the objects created before the invocation. The generalized algorithms for lazy/lazier/lazier# initializations then choose from existing heap objects with version numbers less than or equal to that of the object instead of choosing from existing heap objects that are symbolic. For example, the parameter `wrapper` starts with the context version 0, and the version of the object o pointed by `m` at line 4 is 1. Meanwhile, the returned object from `bar` at line 6 has 2 as its context version. Therefore, the choosing range of `w.myInt` at line 7 includes the object o pointed by `m` because $1 \leq 2$.

3.4 Implementation

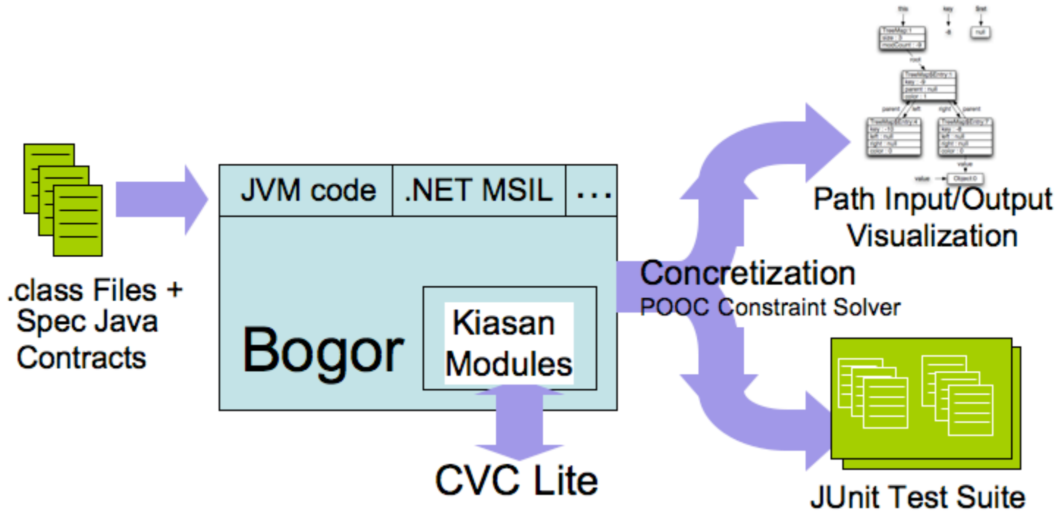


Figure 3.9: *Tool Architecture*

We have implemented Kiasan using the Bogor framework [54, 3]. Figure 3.9 shows the architecture of Kiasan. The prototype uses a specification processor similar to `jmlc` [16] that translates annotated Java source code and embeds the effective contracts in the code. The resulting code is compiled into classes using a standard Java compiler, the classes then translated to Bogor's input language (BIR) extended with Java bytecode instructions modeled as BIR language extensions [3]. Each language extension modeling a Java bytecode is interpreted using the semantics presented in Section 5.2 and enhanced with the versioning techniques described in Section 3.3. The Java bytecode to BIR translation uses the ASM bytecode engineering framework [2], and it virtually generates one atomic transition for each bytecode. During the translation, we close the system as described in Section 3.3. We use CVC Lite [11] as a decision procedure to determine satisfiability of path conditions. For KUnit, we use POOC [57] for solving constraints. KUnit generates a JUnit test case and input/output visualization for each symbolic path that Kiasan explores.

Chapter 4

Optimality Analysis

In Chapter 3, we are concerned with whether lazier initialization is optimal, that is, it generates minimum numbers of cases. In this chapter, we calculate the possible numbers of cases for three types of trees (binary search tree, AVL tree, and red-black tree) and other data structures using a combinatorics technique called *generating functions* [64, 30]. For each data structure, we calculate the numbers of different (non-isomorphic) structures and the numbers of cases after certain operations such as `search/insert/remove`. To our best knowledge, counting cases after certain operations has not been done before whereas previous work only counts the non-siomorphic structures.

Section 4.1 presents the foundation of calculating numbers of cases for the binary search trees. (Note that AVL tree and red-black tree are special cases of binary search tree.) Sections 4.2, 4.3, and 4.4 show the counting of binary search tree, red-black tree, and AVL tree respectively. List merge counting is presented in Section 4.5. Finally, binary heap counting is discussed in Section 4.6.

Acknowledgments

This chapter is an expansion of a paper titled "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs" by Xianghua Deng, Robby, and John Hatcliff to appear in the Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods [25].

4.1 Foundation of Binary Search Tree Counting

This section presents the foundation for counting numbers of non-isomorphic binary search trees [40] (including red-black trees and AVL trees) and the numbers of cases after certain operations such as `insert/remove/search`. The definition of binary search trees is presented in Section 8.2.

Without loss of generality (WLOG), we assume that the elements contained in tree nodes are integers ($\mathbb{Z}$). First we define BST to be the set of all binary search trees and $BST_n = \{ t \in BST \mid height(t) < n \}$ for all $n \in \mathbb{N}$. Then we define two relations.

- $R : BST \times BST$ as

$$t_1 \, R \, t_2 \iff \exists f : \mathbb{Z} \rightharpoonup \mathbb{Z}. \operatorname{dom} f \supseteq elements(t_1) \wedge f \text{ is strictly increasing } \wedge f(t_1) = t_2. \quad (4.1)$$

  where $elements(t)$ returns all the elements of tree t and $f(t)$ substitutes elements of t using f and keeps the structure of t. The relation R is an equivalence relation:

  1. reflexivity, let f be the identity map then we get $t \, R \, t$ for all $t \in BST$.
  2. symmetry, if we have $t_1 \, R \, t_2$ for some f, we need to show $t_2 \, R \, t_1$. By the property of f is strictly increasing, f must be injective. Then we know that f^{-1} is a partial function and strictly increasing. So we get $t_2 \, R \, t_1$ by f^{-1}.
  3. transitivity, if we have $t_1 \, R \, t_2$ and $t_2 \, R \, t_3$, we need to show $t_1 \, R \, t_3$. Suppose f_1 maps t_1 to t_2 and f_2 maps t_2 to t_3. We can define a function $f' : \mathbb{Z} \to \mathbb{Z}$ as $f' = f_2 \circ f_1$ and clearly f' is strictly increasing. Thus we conclude that $t_1 \, R \, t_3$ by f'.

- $R' : (BST \times \mathbb{Z}) \times (BST \times \mathbb{Z})$

$$(t_1, x_1) \, R' \, (t_2, x_2) \iff \exists f : \mathbb{Z} \rightharpoonup \mathbb{Z}. \operatorname{dom} f \supseteq elements(t_1) \cup \{x_1\} \wedge f \text{ is strictly increasing} \wedge f(x_1) = x_2 \wedge f(t_1) = t_2. \quad (4.2)$$

  Similarly, R' is also an equivalence relation.

We want to count two things:

1. $|BST_n/R|$, the number of partitions of binary search trees with heights less than n. Essentially, we count the number of unlabeled binary trees.

2. $|(BST_n \times \mathbb{Z})/R'|$, the number of partitions of pairs of binary search trees with heights less than n and integers. Obviously,

$$(BST_n \times \mathbb{Z})/R' = \biguplus_{T \in BST_n/R} (T \times \mathbb{Z})/R'.$$

Now we proceed to count $|(T \times \mathbb{Z})/R'|$ for $T \in BST_n/R$. We claim that

$$|(T \times \mathbb{Z})/R'| = 2 \times \#nodes(T) + 1,$$

where $\#nodes(T)$ is the number of nodes of any tree in T. Clearly, all the trees in T have the same shape, $\#nodes(T)$ is well-defined. Suppose $\#nodes(T) = k$ and we define a tree $t \in T$ which has elements: $2, 4, \ldots, 2k$. Then define $P = \{ (t, i) \mid 1 \leq i \leq 2k + 1 \}$. Clearly, $P \subset (T \times \mathbb{Z})$. Also it is easy to see that no two $(t, i_1), (t, i_2) \in P$ with $i_1 \neq i_2$ are in the same partition, that is, $\neg(t, i_1)\, R'\, (t, i_2)$. If we can show for all $(t', x) \in (T \times \mathbb{Z})$, $(t', x)\, R'\, p$ for some $p \in P$, then we can conclude $|(T \times \mathbb{Z})/R'| = |P| = 2 \times \#nodes(T) + 1$. Given any $(t', x) \in (T \times \mathbb{Z})$ and the elements in t' are $e_1, e_2, \ldots, e_k$ (in increasing order), since $t', t \in T$, then $t'\, R\, t$, that is, there exists a strictly increasing function f such that $f(t') = t$. Then we know $f(e_i) = 2i$ for all $1 \leq i \leq k$. If $x = e_i$ for some $1 \leq i \leq k$, we get $(t', x)\, R'\, (t, 2i)$ by f. Otherwise, suppose $e_1 < x < e_2$, we define a new function f' as follows:

$$f'(y) = \begin{cases} f(y) & \text{if } f(y) \text{ is defined and } y \geq e_2 \text{ or } e_1 \geq y \\ 3 & \text{if } y = x \\ undefined & \text{otherwise} \end{cases}.$$

Clearly, f' is strictly increasing. The other cases are similar. Therefore, we get $(t', x)\, R'\, (t, 3)$ by f'. We conclude that

$$|(BST_n \times \mathbb{Z})/R'| = \sum_{T \in BST_n/R} 2 \times \#nodes(T) + 1.$$

This number is used to count the number of cases after the `search/insert/remove` operations. The `search/insert/remove` operations are similar:

- these operations take in a tree t and an integer x;
- the most important part of these operations is to find/search the suitable position for x in t. Suppose t has n nodes, there are total $2n + 1$ positions that include n nodes and $n + 1$ NULLS. That is, for any binary search tree t with n nodes,

$$|\{\, [(t, x)]_{R'} \mid x \in \mathbb{Z} \,\}| = 2n + 1.$$

Clearly, $|BST_n/R|$ is the number of non-isomorphic binary search trees and $|(BST_n \times \mathbb{Z})/R'|$ is the number of cases after the `search/insert/remove` operations.

4.2 Counting Binary Search Trees

4.2.1 Counting Numbers of Binary Search Trees BST_n/R

Since we only consider tree shapes but not the labels (elements) of tree nodes, the number of binary search trees with heights less than n is the same as the number of binary trees with heights less than n. Let a_n be the number of binary trees whose heights less than n for $n \geq 0$. We admit the empty tree as a legal binary tree with height -1. Clearly we have $a_0 = 1$ because only the empty tree's height is less than 0. Let consider a_n for $n \geq 1$. Then for each tree of height less than n, either it is empty or non-empty. For a non-empty binary tree, it must have a root. The heights of the left and right subtrees of the root are less than $n - 1$. Therefore, we get

$$a_n = 1 + a_{n-1}^2 \quad n \geq 1 \quad (a_0 = 1). \tag{4.3}$$

We get

$$\begin{aligned}
a_1 &= 1 + a_0^2 = 2 \\
a_2 &= 1 + a_1^2 = 1 + 2^2 = 5 \\
a_3 &= 1 + a_2^2 = 1 + 5^2 = 26 \\
a_4 &= 1 + a_3^2 = 1 + 26^2 = 677 \\
&\vdots
\end{aligned}$$

This sequence grows double exponentially. In fact, Aho [4] showed $a_n = [k^{2^n}]$ where [] is the nearest integer function and $k = 1.502837\ldots$.

4.2.2 Counting $(BST_n \times \mathbb{Z})/R'$

Let $b(m, n)$ be $|\{\, [t]_R \mid t \in BST_n \wedge t \text{ has } m \text{ nodes} \,\}|$, the number of binary trees with m nodes and heights less than n. Clearly $b(0, n) = 1$ for all $n \geq 0$ and $b(m, 0) = 0$ for all $m \geq 1$. Let $c_n = (BST_n \times \mathbb{Z})/R'$. We have $c_n = \sum_{0 \leq i}(2i + 1)b(i, n)$. [1] Define a generating function for $b(m, n)$ as

$$T_n(x) = \sum_{m \geq 0} b(m, n)x^m \quad n \geq 0. \tag{4.4}$$

Note from the definition of $b(m, n)$, we can see clearly that $a_n = T_n(1)$ where a_n is the number of binary trees whose heights are less than n. A non-empty tree with height less than n and $m > 0$ nodes can have a left subtree with i nodes and height less than $n - 1$ and a right subtree with $m - 1 - i$ nodes and height less than $n - 1$ for any $0 \leq i \leq m - 1$. Thus we get

$$b(m, n) = \sum_{i+j=m-1} b(i, n - 1)b(j, n - 1) \quad m > 0, n \geq 1 \tag{4.5}$$

After multiplying x^m to both sides of (4.5) and summing over $1 \leq m \leq \infty$, we get

$$T_n(x) = x(T_{n-1}(x))^2 + 1 \quad n \geq 1. \tag{4.6}$$

[1]This result allows the duplicated resulting trees and corresponds to the stateless case.

Since $b(0,0) = 1$ and $b(m,0) = 0$ for $m > 0$, we have $T_0(x) = 1$. Using recurrence (4.6), we can get $T_1(x) = 1 + x$, $T_2(x) = 1 + x + 2x^2 + x^3$, etc. From the definitions of a_n and $b(m,n)$, we know $a_n = \sum_{m\geq 0} b(m,n)$. Thus $a_n = T_n(1)$ and (4.6) becomes (4.3) with $x = 1$ as expected.

Next, define generating function

$$G_n(x) = \sum_{m\geq 0}(2m+1)b(m,n)x^m \quad n \geq 0. \tag{4.7}$$

Then

$$\begin{aligned} G_n(x) &= \sum_{m\geq 0}(2m+1)b(m,n)x^m \\ &= 2\sum_{m\geq 0} mb(m,n)x^m + \sum_{m\geq 0} b(m,n)x^m \\ &= 2xT_n'(x) + T_n(x), \end{aligned}$$

where $T_n'(x)$ is the derivative of $T_n(x)$. Clearly, $c_n = G_n(1)$. In order to get the closed formula of $G_n(x)$, we need to calculate $T_n'(x)$,

$$T_n'(x) = (x(T_{n-1}(x))^2)' = 2xT_{n-1}(x)T_{n-1}'(x) + (T_{n-1}(x))^2 \quad n \geq 1, T_0'(x) = 0.$$

Thus we get a recurrence relation

$$T_n'(1) = 2T_{n-1}(1)T_{n-1}'(1) + (T_{n-1}(1))^2. \tag{4.8}$$

We know that $T_n(1) = a_n = [k^{2^n}]$. Then the recurrence (4.8) becomes

$$T_n'(1) = 2[k^{2^{n-1}}]T_{n-1}'(1) + [k^{2^{n-1}}]^2. \tag{4.9}$$

Let $b(n) = 2[k^{2^{n-1}}]$ and $c(n) = [k^{2^{n-1}}]^2$. Using the technique in page 18 of [31], multiply both sides of (4.9) by $F(n) = 1/\prod_{j=1}^n b(j)$ and get

$$y_n = y_{n-1} + F(n)c(n),$$

where $y_n = b(n+1)F(n+1)T_n'(1)$. Finally we can get

$$T_n'(1) = \frac{T_0'(1) + \sum_{i=1}^n F(i)c(i)}{b(n+1)F(n+1)}.$$

Now we can substitute in $F(n)$, $b(n)$, and $T_0'(1) = 0$ and get

$$T_n'(1) = \frac{0 + \sum_{i=1}^{n} \frac{1}{\prod_{j=1}^{i} 2[k^{2^{j-1}}]}[k^{2^{i-1}}]^2}{2[k^{2^n}]/\prod_{j=1}^{n+1} 2[k^{2^{j-1}}]} = \sum_{i=1}^{n}[k^{2^{i-1}}]^2 \prod_{j=i+1}^{n} 2[k^{2^{j-1}}].$$

Therefore, we get

$$c_n = G_n(1) = 2T_n'(1) + T_n(1) = 2\sum_{i=1}^{n}[k^{2^{i-1}}]^2 \prod_{j=i+1}^{n} 2[k^{2^{j-1}}] + [k^{2^n}]. \tag{4.10}$$

Now we can calculate the first few terms of c_n:

$$\begin{aligned}
c_0 &= [k^{2^0}] = 1,\\
c_1 &= 2\sum_{i=1}^{1}[k^{2^{i-1}}]^2 \prod_{j=i+1}^{1} 2[k^{2^{j-1}}] + [k^{2^1}] = 2 + 2 = 4,\\
c_2 &= 2\sum_{i=1}^{2}[k^{2^{i-1}}]^2 \prod_{j=i+1}^{2} 2[k^{2^{j-1}}] + [k^{2^2}] = 16 + 5 = 21,\\
c_3 &= 2\sum_{i=1}^{3}[k^{2^{i-1}}]^2 \prod_{j=i+1}^{3} 2[k^{2^{j-1}}] + [k^{2^3}] = 210 + 26 = 236,\\
c_4 &= 2\sum_{i=1}^{4}[k^{2^{i-1}}]^2 \prod_{j=i+1}^{4} 2[k^{2^{j-1}}] + [k^{2^4}] = 2 \times 6136 + 667 = 12939,\\
&\vdots
\end{aligned}$$

Numbers of non-isomorphic binary search trees after insert operation Now we only consider the `insert` operation and want to find out the number of non-isomorphic binary search trees after insertion,

$$f_n = |\{\, \texttt{insert}(t, x)/R \mid t \in BST_n \wedge x \in \mathbb{Z} \,\}|, \tag{4.11}$$

where `insert`(t, x) is the binary search tree after inserting x into tree t. The above calculation of c_n trees contain a lot of duplications. For example, the empty tree inserted with element 1 will end up with the same tree as a tree with a single node 1 inserted with element 1. Clearly the set of non-isomorphic binary search trees after insertion consists of all the input binary trees except the empty tree and the set of binary trees with one node of depth n. Let e_h be the total number of binary trees (after insertion) with heights h. Since after insertion, the resulting tree can not be empty, we have

$$f_h = a_h + e_h - 1 \quad h > 1, \tag{4.12}$$

and $f_0 = 1$.

In order to calculate e_h, we need to count the number of binary trees with one node of depth n. Define $d(h, l)$ as the number of binary trees with heights less than h and having l nodes with depth $h - 1$ for $h \geq 0, l \geq 0$. Then $d(0, 0) = 1, d(0, n) = 0$, for $n > 0$, $d(1, 1) = 1$, and $d(h, 0) = a_{h-1}$. Similar to (4.5), since each node of depth $h-1$ can have a left or right new child, $e_h = \sum_{l\geq 0} 2l{\cdot}d(h, l)$ for $h > 1$ and $e_0 = 1, e_1 = 2$. Then for $h > 1$, we have

$$d(h, l) = \sum_{i+j=l} d(h - 1, i)d(h - 1, j) \quad h > 1.$$

Define a generating function for $d(h, l)$ as

$$F_h(x) = \sum_{i\geq 0} d(h, i)x^i.$$

Then we get $F_h(x) = (F_{h-1}(x))^2$ for $h > 1$ and $F_0(x) = 1, F_1(x) = 1 + x$. Since $e_h = \sum_{l\geq 0} 2l \cdot d(h, l)$ for $h > 1$, $e_h = 2F_h'(1) = 2\left((1 + x)^{2^{h-1}}\right)'(1) = 2^h(1 + 1) = 2^{h+1}$ for $h > 1$. Thus

$$\begin{aligned} f_0 &= 1 + 2 - 1 = 1, \\ f_1 &= 2 + 2 - 1 = 3, \\ f_2 &= 5 + 8 - 1 = 12, \\ f_3 &= 26 + 16 - 1 = 41, \\ f_4 &= 677 + 2^5 - 1 = 708, \\ &\vdots \end{aligned}$$

4.3 Counting Red-black Trees

A red-black tree[2] [63] is a balanced binary search tree with an additional field “color” in each node. We will denote RBT as the set of red-black trees. Similarly, we define RBT_n as the set of

[2]Definition of red-black tree is presented in Section 8.3.

red-black trees with heights less than or equal to n. In this section, we let red-black trees to have NULL leaf nodes which have no elements. We admit the empty tree (NULL) as a legal red black tree with height 0 and black height 0.

4.3.1 Counting Number of Red-black Trees RBT_n/R

Define

$$a(n,k) = |\{\, t \mid t \in RBT_n \wedge blackheight(t) = k \,\}/R| \tag{4.13}$$

as the number of non-isomorphic red-black trees with heights at most n and black heights k. Clearly we have $a(0,0) = 1$ for only the empty tree with height 0 and black height 0 and $a(n,k) = 0$ for $k > n$. If $k = 0$, the only legal red black tree is the empty tree NULL. Thus $a(n,0) = 1$ for all $n \geq 0$. Let us consider $a(n,k)$ for $k \geq 1$ and $n \geq k$. By the properties of red-black trees, the root of any non-empty red-black tree has to be black. There are four cases according to the colors of the children of the root as shown in Figure 4.1:

1. both the left and right children of the root node are black as shown in Figure 4.1(a). Then two subtrees have heights less than $n-1$ and black heights $k-1$.

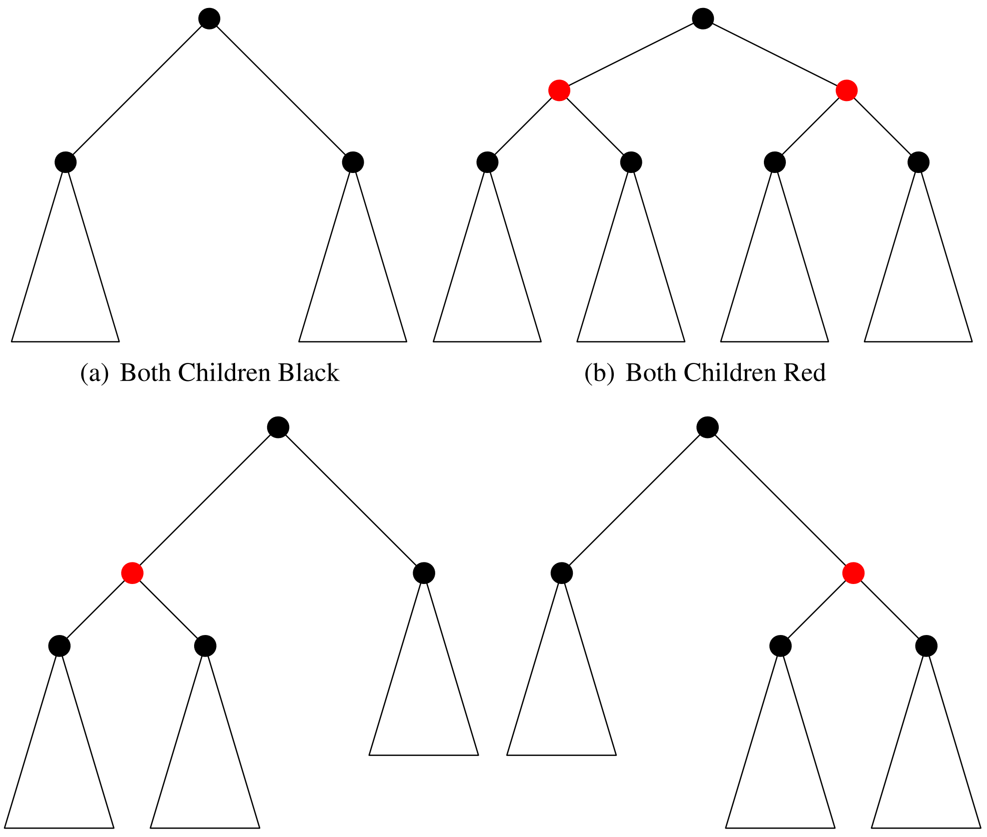
2. the left child is black but the right child is red as shown in Figure 4.1(d). Then two subtrees of the left child have heights less than or equal to $n-2$ and black heights $k-1$. Right child of the root has height less than $n-1$ and black height $k-1$.

3. the right child is red but the left child is black as shown in Figure 4.1(c). It is symmetric to the black-red case.

4. both the left and right children are red as shown in Figure 4.1(b). Four grandchildren of the root have heights less than or equal to $n-2$ and black heights $k-1$.

Therefore, we get

$$\begin{aligned} a(n,k) &= a(n-1,k-1)^2 + 2a(n-1,k-1)a(n-2,k-1)^2 + a(n-2,k-1)^4 \\ &= [a(n-1,k-1) + a(n-2,k-1)^2]^2, \quad n \geq 1, k \geq 1. \end{aligned}$$


(a) Both Children Black (b) Both Children Red

(c) Left Child Red and Right Child Black (d) Left Child Black and Right Child Red

Figure 4.1: *Red-black Tree Counting Cases*

We let $a(n,k) = 0$ for $n < 0$. Then we get $a(1,1) = 1$.

$$\begin{aligned}
a(2,1) &= [a(1,0) + a(0,0)^2]^2 = (1 + 1^2)^2 = 4.\\
a(2,2) &= [a(1,1) + a(0,1)^2]^2 = (1 + 0)^2 = 1.\\
a(3,1) &= [a(2,0) + a(1,0)^2]^2 = (1 + 1^2)^2 = 4.\\
a(3,2) &= [a(2,1) + a(1,1)^2]^2 = (4 + 1^2)^2 = 25.\\
a(3,3) &= [a(2,2) + a(1,2)^2]^2 = (1 + 0^2)^2 = 1.\\
a(4,1) &= [a(3,0) + a(2,0)^2]^2 = (1 + 1^2)^2 = 4.\\
a(4,2) &= [a(3,1) + a(2,1)^2]^2 = (4 + 4^2)^2 = 400.\\
a(4,3) &= [a(3,2) + a(2,2)^2]^2 = (25 + 1^2)^2 = 676.\\
a(4,4) &= [a(3,3) + a(2,3)^2]^2 = (1 + 0^2)^2 = 1.\\
&\vdots
\end{aligned}$$

Let $b_n = |RBT_n/R|$ be the number of non-isomorphic red-black trees with heights less than or equal to n. Clearly $b_n = \sum_{k=0}^{n} a(n,k)$. Thus we get

$$\begin{aligned}
b_0 &= a(0,0) = 1,\\
b_1 &= a(1,0) + a(1,1) = 2,\\
b_2 &= a(2,0) + a(2,1) + a(2,2) = 1 + 4 + 1 = 6,\\
b_3 &= a(3,0) + a(3,1) + a(3,2) + a(3,3) = 1 + 4 + 25 + 1 = 31,\\
b_4 &= a(4,0) + a(4,1) + a(4,2) + a(4,3) + a(4,4) = 1 + 4 + 400 + 676 + 1 = 1082,\\
&\vdots
\end{aligned}$$

4.3.2 Counting $(RBT_n \times \mathbb{Z})/R'$

We will first count the numbers of non-isomorphic red-black trees indexed by heights. Define

$$f(n,h,k) = |\{\, t \in RBT_h \mid blackheight(t) = k \wedge leaf(t) = n \,\}/R|, \tag{4.14}$$

the number of non-isomorphic red-black trees with n leaf nodes (NULLs) and heights less than or equal to h and black heights equal to k. So we have

$$\begin{aligned} f(n,h,k) = \sum_{i+j=n} f(i,h-1,b-1)a(j,h-1,b-1) + \sum_{i+j+k=n} f(i,h-2,b-1)f(j,h-2,b-1)f(k,h-1,b-1) \\ + \sum_{i+j+k=n} f(i,h-1,b-1)f(j,h-2,b-1)f(k,h-2,b-1) \\ + \sum_{i+j+k+l=n} a(i,h-2,b-1)a(j,h-2,b-1)a(k,h-2,b-1)a(l,h-2,b-1), \end{aligned}$$

for $n > 0$. Clearly, we have $f(1,0,0) = 1$ and $f(1,h,k) = 0$ for $(h,k) \neq (0,0)$ and $f(n,h,k) = 0$ for $h < 0$ or $k < 0$. Define a generating function $F_{h,b}(x) = \sum_{n=1}^{\infty} a(n,h,b)x^n$ for $k \geq 0$. Then we get

$$F_{h,b}(x) = F^2_{h-1,b-1}(x) + 2F^2_{h-2,b-1}(x)F_{h-1,b-1}(x) + F^4_{h-2,b-1}(x) = (F_{h-1,b-1}(x) + F^2_{h-2,b-1}(x))^2.$$

The boundary condition is

$$F_{h,0}(x) = \begin{cases} x, & \text{if } h \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

We can get $F_{1,1}(x) = x^2, F_{2,1}(x) = (x+x^2)^2, F_{2,2}(x) = x^4, F_{3,1}(x) = (x+x^2)^2, F_{3,2}(x) = ((x+x^2)^2 + x^4)^2, F_{3,3}(x) = x^8, F_{4,1}(x) + (x+x^2)^2 = x^4 + 2x^3 + x^2,$

$$\begin{aligned} F_{4,2}(x) = (F_{3,1}(x) + F_{2,1}(x)^2)^2 = [(x+x^2)^2 + (x+x^2)^4]^2 = x^4 + 4x^5 + 8x^6 + 16x^7 + \\ 32x^8 + 48x^9 + 58x^{10} + 68x^{11} + 72x^{12} + 56x^{13} + 28x^{14} + 8x^{15} + x^{16}, \end{aligned}$$

$$\begin{aligned} F_{4,3}(x) = (F_{3,2}(x) + F_{2,2}(x)^2)^2 = [((x+x^2)^2 + x^4)^2 + x^8]^2 = x^8 + 8x^9 + 32x^{10} + \\ 80x^{11} + 138x^{12} + 168x^{13} + 144x^{14} + 80x^{15} + 25x^{16}, \end{aligned}$$

and $F_{4,4}(x) = x^{16}$. Define $G_h(x) = \sum_{i=0}^{h} F_{h,i}(x)$. So $[x^n]G_h(x)$ is the number of non-isomorphic red-black trees with $n-1$ nodes [3] and heights less than or equal to h. Let compute $G_h(x) =$

[3]This is because for a n node binary tree, it has $n+1$ NULL leaves [56].

$g(h,0) + g(h,1)x + g(h,2)x^2 + \cdots$:

$$\begin{aligned}
G_1(x) &= x + x^2, \\
G_2(x) &= 2x^4 + 2x^3 + x^2 + x, \\
G_3(x) &= x^8 + ((x+x^2)^2 + x^4)^2 + (x+x^2)^2 + x \\
&= 5x^8 + 8x^7 + 8x^6 + 4x^5 + 2x^4 + 2x^3 + x^2 + x, \\
G_4(x) &= 27x^{16} + 88x^{15} + 172x^{14} + 224x^{13} + 210x^{12} + 148x^{11} + \\
&\quad 90x^{10} + 56x^9 + 33x^8 + 16x^7 + 8x^6 + 4x^5 + 2x^4 + 2x^3 + x^2 + x \\
&\vdots
\end{aligned}$$

Now we will compute $p_h = |(RBT_h \times \mathbb{Z})/R'|$, the total number of cases after the `insert` operation for red-black trees with height less than or equal to h is

$$p_h = \sum_{i \geq 0} (2i-1) g(h,i).$$

Then we have $p_h = 2G'(1) - G(1)$.

$$\begin{aligned}
p_1 &= 2 \times 3 - 2 = 4, \\
p_2 &= 2 \times 17 - 6 = 28, \\
p_3 &= 2 \times (40 + 56 + 48 + 20 + 8 + 6 + 2 + 1) - 31 = 331, \\
P_4 &= 2 \times (27 \cdot 16 + 88 \cdot 15 + 172 \cdot 14 + 224 \cdot 13 + 210 \cdot 12 + 148 \cdot 11 + 90 \cdot 10 + 56 \cdot 9 + 33 \cdot 8 + 16 \cdot 7 \\
&\quad + 8 \cdot 6 + 4 \cdot 5 + 2 \cdot 4 + 2 \cdot 3 + 2 + 1) - 1082 = 2 \cdot 13085 - 1082 = 25088, \\
&\vdots
\end{aligned}$$

4.4 Counting AVL Trees

AVL Tree [40] is another balanced binary search tree. The structure constraint is that for any node in the tree, the heights of its left subtree and right subtree differ at most by 1. Similar to red-black tree, we treat NULLS as legal nodes.

We define AVL be the set of all AVL tree and $AVL_h = \{ t \in AVL \mid height(t) = h \}$ for $h \in \mathbb{N}$. So AVL_0 is a singleton that only contains the empty tree NULL.

4.4.1 Counting Numbers of AVL Trees AVL_h/R

Define $a_h = |AVL_h/R|$, the number of non-isomorphic AVL trees with heights equal to h. For an AVL tree with height h greater than 0, there are three cases according to the heights of its left and right subtrees:

- the heights of the left and right subtrees are the same. So the heights of the left and right subtrees must be $h - 1$.
- the height of the left subtree is greater than the height of the right subtree by one. So the height of the left subtree is $h - 1$ and the height of the right subtree is $h - 2$.
- the height of the left subtree is less than the height of the right subtree by one. So the height of the left subtree is $h - 2$ and the height of the right subtree is $h - 1$.

So we get

$$a_h = a_{h-1}^2 + 2a_{h-1}a_{h-2} \quad h > 0. \tag{4.15}$$

The boundary condition is $a_0 = 1$. Thus we have

$$\begin{aligned}
a_1 &= a_0^2 = 1, \\
a_2 &= a_1^2 + 2a_0a_1 = 3, \\
a_3 &= a_2^2 + 2a_2a_1 = 15, \\
a_4 &= a_3^2 + 2a_3a_2 = 15^2 + 2 \times 15 \times 3 = 315, \\
&\vdots
\end{aligned}$$

Note, [40] shows that

$$a_h = \lfloor \theta^{2^h} \rfloor - \lfloor \theta^{2^{h-1}} \rfloor + \lfloor \theta^{2^{h-2}} \rfloor - \cdots + (-1)^h \lfloor \theta^{2^0} \rfloor,$$

where $\theta \approx 1.43687$.

4.4.2 Counting $(AVL_h \times \mathbb{Z})/R'$

Define $b(h,n) = |\{\, [t]_R \mid t \in AVL_h \wedge t \text{ has } n \text{ nodes} \,\}|$, the number of non-isomorphic AVL trees with n nodes and height h. Then we have

$$b(h,n) = \sum_{i+j=n-1} b(h-1,i)b(h-1,j) + 2\sum_{i+j=n-1} b(h-1,i)b(h-2,j) \quad h > 0, n \geq 1. \tag{4.16}$$

Clearly, we have $b(0,0) = 1$ and $b(0,n) = 0$ for $n > 0$. Define a sequence of generating functions, for $h \geq 0$,

$$H_h(x) = \sum_{i \geq 0} b(h,i)x^i. \tag{4.17}$$

We have $H_0(x) = 1$. We can multiply (4.16) by x^n and sum over $1 \leq n \leq \infty$ and get

$$H_h(x) = xH_{h-1}^2(x) + 2xH_{i-1}(x)H_{i-2}(x) \quad h > 0. \tag{4.18}$$

So we get

$$\begin{aligned}
H_1(x) &= x, \\
H_2(x) &= x \times x^2 + 2x \times x = x^3 + 2x^2, \\
H_3(x) &= 4x^4 + 6x^5 + 4x^6 + x^7, \\
&\vdots
\end{aligned}$$

Now we will compute $c_h = |(AVL_h \times \mathbb{Z})/R'|$, the total number of cases after `insert` operation for AVL trees with height equal to h,

$$c_h = \sum_{i \geq 0} (2i+1)b(h,i).$$

Then we have $c_h = 2H_h'(1) + H_h(1)$. From (4.18), we get

$$H_h'(1) = H_{h-1}^2(1) + H_{h-1}'(1)H_{h-1}(1) + 2H_{n-1}(1)H_{n-2}(1) + 2H_{h-1}'(1)H_{h-2}(1) + 2H_{h-1}(1)H_{h-2}'(1).$$

Then we get $H'_0(1) = 0, H'_1(1) = 1, H'_2(1) = 7, H'_3(1) = 77, \ldots$. Therefore,

$$\begin{aligned}
c_0 &= 2 \times 0 + 1 = 1, \\
c_1 &= 2 \times 1 + 1 = 3, \\
c_2 &= 2 \times 7 + 3 = 17, \\
c_3 &= 2 \times 77 + 15 = 169, \\
&\vdots
\end{aligned}$$

4.5 Counting List with Merge Operation

The `merge` [40] operation takes two sorted lists and merges them into one sorted list. Since `merge` takes two lists and each list has length less than k, the total number of cases before the operation is $k \times k = k^2$.

Number of cases after the `merge` operation equals to

$$\begin{aligned}
b_k &= \sum_{0 \le m,n < k} \binom{m+n}{n} \\
&= \sum_{0 \le m < k} \sum_{0 \le n < k} \binom{m+n}{n} \\
&= \sum_{0 \le m < k} \binom{m+k}{k-1} \\
&= \binom{k-1+1+k}{k-1+1} - 1 = \binom{2k}{k} - 1.
\end{aligned}$$

Number of cases after the operation if postcondition is strictly ascending Now we need to consider additional error cases: two elements in two lists are equal. Let the two input lists to be $[c_1, c_2, \ldots, c_n]$ and $[d_1, d_2, \ldots, d_m]$. If all the c_i, d_j are distinct, there are b_k cases after merging. Otherwise, suppose that c_i and d_j are the first pair that equals, so we have to merge sub-lists before c_i, d_j ($[c_1, c_2, \ldots, c_{i-1}]$ and $[d_1, d_2, \ldots, d_{j-1}]$) and after the two elements ($[c_{i+1}, c_{i+2}, \ldots, c_n]$

and $[d_{j+1}, d_{j+2}, \ldots, d_m]$). Thus there are

$$\sum_{0\le m,n<k} \sum_{1\le i\le n, 1\le j\le m} \binom{i-1+j-1}{i-1} \times \binom{n-i+m-j}{n-i}$$

extra cases.

4.6 Counting Binary Heap

Binary Heap [63] is a special kind of binary tree with two properties:

- *structure property* a heap is a *complete binary tree*, that is, a binary tree that is completely filled with possible exception of the bottom level which is filled from left to right;
- *heap-order property* any node is smaller than its children.

The number of non-isomorphic binary heaps with n nodes is 1 because of the structure property.

The number of cases after `deleteMin` operation is $n-1$ for a n node input binary heap because the last node can be put in any of the previous $1..n-1$ position. Once the position of the last node is decided, the other nodes are moved deterministically.

The number of cases after `insert` operation for a n node binary heap is $\lfloor \log(n+1) \rfloor + 1$ because the inserted element can be put in any place in the path from the root to the last element and $\lfloor \log(n+1) \rfloor$ is the number of nodes in the shortest path from the root to a leaf.

Chapter 5

Formalization of Symbolic Executions in Kiasan

In this chapter, we formalize the basic (non-compositional) symbolic executions in Kiasan and prove the symbolic executions with lazy, lazier, and lazier# initialization are relatively sound and complete with respect to a JVM concrete execution operational semantics. Section 5.1 defines some substitution functions that will be used throughout this chapter and Chapter 7. Operational semantics of symbolic executions with lazy, lazier, and lazier# initialization and concrete JVM operational semantics are presented in Section 5.2. Finally, Section 5.3 shows the proof of relative soundness and completeness.

Acknowledgments

This chapter is partially adapted from two papers: first titled "Bogor/Kiasan: A *k*-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems" by Xianghua Deng, Jooyong Lee, and Robby that appeared in the Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering [23]; second titled "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs" by Xianghua Deng, Robby, and John Hatcliff to be appeared in the Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods [25].

5.1 Substitution Functions

First we will define some substitution functions. Assume that D, D' are some domains and $\mathrm{Seq}(D)$ is the set of all sequences of elements in D:

- a substitution function, $\mathit{sub} : D \times (D \rightharpoonup D) \to D$ as
$$\mathit{sub}(d, g) = \begin{cases} g(d) & \text{if } d \in \operatorname{dom} g; \\ d & \text{otherwise.} \end{cases}$$

- a function substitution function, $\mathit{sub\text{-}fun} : (D' \rightharpoonup D) \times (D \rightharpoonup D) \to (D' \rightharpoonup D)$ as $\mathit{sub\text{-}fun}(f, g) = f'$ where $\operatorname{dom} f = \operatorname{dom} f'$ and $\forall d \in \operatorname{dom} f . f'(d) = \mathit{sub}(f(d), g)$.

- a one-step function substitution function, $\mathit{sub\text{-}fun}_1 : (D' \rightharpoonup D) \times D \times D \to (D' \rightharpoonup D)$ as $\mathit{sub\text{-}fun}_1(f, d, d') = \mathit{sub\text{-}fun}(f, \{(d, d')\})$.

- a sequence substitution function: $\mathit{sub\text{-}seq} : \mathrm{Seq}(D) \times (D \rightharpoonup D) \to \mathrm{Seq}(D)$ as $\mathit{sub\text{-}seq}(\mathit{nil}, g) = \mathit{nil}$ and $\mathit{sub\text{-}seq}(d :: q, g) = \mathit{sub}(d, g) :: \mathit{sub\text{-}seq}(q, g)$.

- an one-step sequence substitution function: $\mathit{sub\text{-}seq}_1 : \mathrm{Seq}(D) \times D \times D \to \mathrm{Seq}(D)$ as $\mathit{sub\text{-}seq}_1(q, d, d') = \mathit{sub\text{-}seq}(q, \{(d, d')\})$.

- a functional substitution function $\mathit{sub\text{-}fun2} : (D'' \rightharpoonup D' \rightharpoonup D) \times (D \rightharpoonup D) \to (D'' \rightharpoonup D' \rightharpoonup D)$ as $\mathit{sub\text{-}fun2}(f, g) = f'$ where $\operatorname{dom} f = \operatorname{dom} f'$ and $\forall d'' \in \operatorname{dom} f . f'(d'') = \mathit{sub\text{-}fun}(f(d''), g)$.

- a one-step functional substitution function $\mathit{sub\text{-}fun2}_1 : (D'' \rightharpoonup D' \rightharpoonup D) \times D \times D \to (D'' \rightharpoonup D' \rightharpoonup D)$ as $\mathit{sub\text{-}fun2}_1(f, d, d') = \mathit{sub\text{-}fun2}(f, \{(d, d')\})$.

Then we introduce some simple properties of the substitution functions:

Lemma 1. *Suppose a partial function $g : D \rightharpoonup D$ for some domain D satisfies $\operatorname{dom} g \cap \operatorname{ran} g = \emptyset$. Then for any $(d, d') \in g$ and function $f : D' \rightharpoonup D$, sequence $q : Seq(D)$, and funtional $f^{ho} : D'' \rightharpoonup D' \rightharpoonup D$, we have $\mathit{sub\text{-}fun}(f, g) = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}_1(f, d, d'), g)$, $\mathit{sub\text{-}seq}(q, g) = \mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}_1(q, d, d'), g)$, $\mathit{sub\text{-}fun2}(f^{ho}, g) = \mathit{sub\text{-}seq}(\mathit{sub\text{-}fun2}_1(f^{go}, d, d'), g)$.*

Lemma 2. *if R is the range of $f : D' \rightharpoonup D$, the set of elements in a sequence $q : Seq(D)$ or the second range of $f^{ho} : D'' \rightharpoonup D' \rightharpoonup D$, then for any $g : D \rightharpoonup D$, sub-fun(f, g) = sub-fun$(f, g \mid_{R \cap \mathrm{dom}\, g})$, sub-seq$(q, g)$ = sub-seq$(q, g \mid_{R \cap \mathrm{dom}\, g})$, sub-fun2$(f^{ho}, g)$ = sub-fun2$(f^{ho}, g \mid_{R \cap \mathrm{dom}\, g})$.*

5.2 Operational Semantics

This section presents the formal operational semantics of Kiasan's symbolic executions with lazy (SEL), lazier (SELA), and lazier# initialization (SELB), as well as a concrete execution semantics for JVM bytecode instructions.

5.2.1 Operational Semantics of Symbolic Execution with Lazy Initialization

We will discuss the symbolic execution with lazy initialization operational semantics of JVM bytecode with additional two instructions, `assume` and `assert`. First, the semantic domains are introduced. Then some auxiliary functions that facilitate the definition of semantic rules are defined. Finally the semantic rules for bytecode instructions and `assume/assert` are presented.

Semantic Domains

The semantic/syntactical domains are listed as following:

- the set of primitive types, $\mathbf{Types}_{prim}$, consisting of INT, CHAR, etc.,
- the set of array types, $\mathbf{Types}_{array}$,
- the set of record types, $\mathbf{Types}_{record}$,
- the set of symbolic types, **SymTypes**,
- the set of non-primitive types, $\mathbf{Types}_{non-prim} = \mathbf{Types}_{record} \uplus$[1] $\mathbf{Types}_{array} \uplus \mathbf{SymTypes}$,
- the set of all types, $\mathbf{Types} = \mathbf{Types}_{prim} \uplus \mathbf{Types}_{non-prim}$,
- the set of program counters, **PCs**

[1]$\uplus$ denotes disjoint union.

- the set of boolean expressions, Φ,
- the set of locations, **Locs**,
- the set of natural numbers, $\mathbb{N}$,
- the set of constants, **Consts**, including $\mathbb{N}$, True, False, null, etc.,
- the set of fields, **Fields**, including len, def, conc, etc.,
- the set of integer symbols, $\mathbf{Symbols}_{\mathrm{INT}}$,
- the set of primitive symbols, $\mathbf{Symbols}_{prim}$, including $\mathbf{Symbols}_{\mathrm{INT}}$,
- the set of values, $\mathbf{Values} = \mathbf{Consts} \uplus \mathbf{Locs} \uplus \mathbf{Symbols}_{prim}$,
- the set of indexes, $\mathbf{Indexes} = \mathbf{Fields} \uplus \mathbb{N} \uplus \mathbf{Symbols}_{\mathrm{INT}}$,
- the set of non-primitive symbols, $\mathbf{Symbols}_{non-prim} = \{\, X^{m,n}_{\tau} \mid X^{m,n}_{\tau} : \mathbf{Indexes} \rightharpoonup \mathbf{Values} \,\}$,
- the set of symbols, $\mathbf{Symbols} = \mathbf{Symbols}_{prim} \uplus \mathbf{Symbols}_{non-prim}$,
- the set of globals, $\mathbf{Globals} = \{\, g \mid g : \mathbf{Fields} \rightharpoonup \mathbf{Values} \,\}$,
- the set of operand stacks, $\mathbf{Stacks} = \{\, \omega \mid \omega : \mathrm{Seq}(\mathbf{Values}) \,\}$, all sequences of values,
- the set of locals, $\mathbf{Locals} = \{\, l \mid l : \mathbb{N} \rightharpoonup \mathbf{Values} \,\}$,
- the set of heaps, $\mathbf{Heaps} = \left\{ h \mid h : \mathbf{Locs} \rightharpoonup \mathbf{Symbols}_{non-prim} \right\}$,
- the set of bytecode instruction with additional `assert` and `assume` instructions, **Instrs**,

We follow Java type system in the semantic domains: we use $\mathbf{Types}_{prim}$ to model the primitive types and $\mathbf{Types}_{non-prim}$ for the reference types which are divided into object types ($\mathbf{Types}_{record}$), array types ($\mathbf{Types}_{array}$), and symbolic types (**SymTypes**). **SymTypes** is used to model the variable real types of the non-primitive input parameters and global fields.[2] **PCs** denotes the set of

[2]In fact, all the non-primitive symbolic objects are created with symbolic types.

program counters or indexes of code arrays. A special program counter, EOF, is introduced to indicate that the end of code array is reached and execution stops. Similar to types, **Symbols** are divided into two types: primitive symbols, such as symbolic integers, symbolic floats, etc.; and reference symbols including symbolic objects and symbolic arrays. Concrete values are modeled by the **Consts** domain. For simplicity, we unify concrete objects and all symbolic values into the **Symbols** domain. Each member of $\mathbf{Symbols}_{non-prim}$ domain, $X_\tau^{m,n}$, has three properties (we often omit properties when they are not important/applicable): τ is the type of the symbol, m is the object field or array element expansion bound, and n is the number of array element bound. (We will discuss the difference between m and n for arrays at the end of this section.) And each non-primitive symbol, $X_\tau^{m,n}$, is modeled as a partial mapping from its fields to values. Each primitive symbol X_τ or field f_τ also has a property of its type τ. Since arrays are also modeled by **Symbols**, the domain (**Indexes**) of the partial mapping of array X includes natural numbers and symbolic integers. Concrete objects created during the execution are represented as non-primitive symbols too, but their fields are all initialized (see the *new-obj* auxiliary function). On the other hand, fields of symbolic objects may have not been initialized (initially created using the *new-sym* function). Fields of the array include indexes and length, LEN, (which is always defined). Symbolic arrays and concrete arrays are created using the *new-sarr* and *new-arr* functions respectively. **Locs** represents the set of addresses in the heap.

State Since we only consider single threaded programs modularly (one method at a time), we represent a SEL state with only one stack frame element (the stack frame element of the method being analyzed). A state is represented as a tuple of global variables, program counter, locals, operand stack, and heap following the JVM specification [43]; we add path condition ϕ (as a conjunctive-set of formulas) as another state component. So the definition of the set of SEL states is :

$$\Sigma_s = \mathbf{Globals} \times \mathbf{PCs} \times \mathbf{Locals} \times \mathbf{Stacks} \times \mathbf{Heaps} \times \Phi$$

and we let σ ranges over Σ_s.

We will follow the convention that

- τ ranges over types, **Types**,
- pc ranges over program counters, **PCs**,
- ϕ ranges over boolean expressions, Φ,
- i and j range over locations, **Locs**,
- m, n, and k range over natural numbers, $\mathbb{N}$,
- c and d range over constants, **Consts**,
- f ranges over fields, **Fields**,
- X, Y, and Z range over symbols, **Symbols**,
- v ranges over values, **Values**,
- ι ranges over indexes, **Indexes**,
- g ranges over globals, **Globals**,
- ω ranges over operand stacks, **Stacks**,
- l ranges over locals, **Locals**,

The meta-variables used to range over the semantic domains may be primed or subscripted.

Auxiliary Functions

We define some auxiliary functions to facilitate the definition of operational semantics:

- default value function, $\mathit{default} : \textbf{Types} \to \textbf{Values}$ as $\lambda\tau.v$, where v is the default value of τ;
- fields of a type function, $\mathit{fields} : \textbf{Types} \to \mathcal{P}(\textbf{Fields})$ as $\lambda\tau.\{f_{\tau'} \mid f_{\tau'} \text{ is a field in } \tau\}$;
- subtype function, $\tau' <: \tau : \textbf{Types} \times \textbf{Types} \to \mathit{Boolean}$ as τ' is a subtype of τ (reflexive);

- defined integral indexes of a non-primitive symbol function, *acc-idx* : $\textbf{Symbols}_{non-prim} \rightarrow \mathcal{P}(\mathbb{N} \cup \textbf{Symbols}_{\text{INT}})$ as $\lambda X. \{ \iota \in \mathbb{N} \cup \textbf{Symbols}_{\text{INT}} \mid X(\iota)\downarrow \}$;

- locations that map to symbolic objects function, *collect* : $\textbf{Heaps} \rightarrow \mathcal{P}(\textbf{Locs})$ as $\lambda h. \{ i \mid h(i)(\textsc{conc}) \uparrow \}$;

- the set of all symbols in a state function, *symbols* : $\Sigma_s \rightarrow \mathcal{P}(\textbf{Symbols})$ as $\lambda\sigma. \{ X \mid X \text{ appears in } \sigma \}$;

- new primitive symbol function, *new-prim-sym* : $\textbf{Types}_{prim} \times \mathcal{P}(\textbf{Symbols}) \rightarrow \textbf{Symbols}_{prim}$ as $\lambda(\tau, ss).X_\tau, X \notin ss$;

- new symbolic type function, *new-sym-type* : $\mathcal{P}(\textbf{Symbols}) \rightarrow \textbf{SymTypes}$ as $\lambda ss.\tau$ s.t. $\tau \in \textbf{SymTypes}$ and τ does not appear in ss;

- new array type function, *array-type* : $\textbf{Types} \rightarrow \textbf{Types}_{array}$ as $\lambda\tau.\tau'$, where τ' is the array type of element type τ;

- new symbolic record function, *new-sym* : $\mathcal{P}(\textbf{Symbols}) \times \mathbb{N} \times \mathbb{N} \rightarrow \textbf{Symbols}_{non-prim}$ as $\lambda(ss, m, n).X_\tau^{m,n}$, s.t. $X \notin ss \wedge \tau = \textit{new-sym-type}(ss) \wedge \forall\iota \in \textbf{Indexes}.X(\iota)\uparrow$;

- new symbolic array function, *new-sarr* : $\mathcal{P}(\textbf{Symbols}) \times \mathbb{N} \times \mathbb{N} \rightarrow \textbf{Symbols}$ as $\lambda(ss, m, n).\textit{new-sym}(ss \cup \{X\}, m, n)[\textsc{len} \mapsto X]$ where $X = \textit{new-prim-sym}(\textsc{int}, ss)$;

- new concrete object function, *new-obj* : $\mathcal{P}(\textbf{Symbols}) \times \textbf{Types}_{record} \rightarrow \textbf{Symbols}$ as $\lambda(ss, \tau).X_\tau^{0,0}$, s.t. $X \notin ss \wedge \forall f_{\tau'} \in \textit{fields}(\tau).X(f_{\tau'}) = \textit{default}(\tau')$;

- new concrete array function, *new-arr* : $\mathcal{P}(\textbf{Symbols}) \times \textbf{Types} \times (\mathbb{N} \uplus \textbf{Symbols}_{\text{INT}}) \times \mathbb{N} \rightarrow \textbf{Symbols}$ as $\lambda(ss, \tau, v, n).X_{\tau'}^{0,n}, X \notin ss \wedge \tau' = \textit{array-type}(\tau) \wedge \operatorname{dom} X = \{\textsc{def}, \textsc{len}, \textsc{conc}\} \wedge X(\textsc{def}) = \textit{default}(\tau) \wedge X(\textsc{len}) = v$.

Semantic Rules

Given an array of instructions, we define a function *code* : **PCs** $\rightharpoonup$ **Instrs** which takes a program counter and returns the corresponding instruction that is pointed to by the input program counter.

Operational semantic rules are in the format of

$$\frac{premises}{\sigma \Rightarrow_S \sigma_1[\| \ \sigma_2] \mid \textsc{Exception}, \sigma' | \textsc{Error}, \sigma''}$$

that shows how a state is changed by one bytecode instruction to multiple normal states, an exception, or an error due to non-determinism. More specifically, given a state σ, if *pre* is satisfied, after executing the instruction pointed by the program counter component of σ, the resulting state is σ_1; or nondeterministically σ_1 or σ_2; or an EXCEPTION with a state σ'; or ERROR with a state σ''. Exceptions are handled the same way as JVM specification [43] does. If an error occurs, then the program stops. For simplicity, we assume that garbage collection is performed after each transition. Moreover, we terminate silently when a state's path condition becomes unsatisfiable.

Each SEL semantic rule name is the format of xxxx#-S where xxxx is the instruction name and since there may be multiple rules for one instruction, we use a number # (from 1 to n) to distinguish the rules for the same instruction. Due to limit of space, we only present semantics for some representative JVM instructions and the instructions are divided into following categories:

- Arithmetic instructions: Instruction `iadd` adds two integers from the top of the stack and puts the result back into the stack. The semantics of `iadd` is represented by the rule IADD-S. A fresh symbolic integer is introduced as the result and a constraint is added to the path condition stating that the fresh symbolic integer equals to the sum of the two operands.

- Object creation and manipulation instructions: `new` τ, `getfield` f, `putfield` f, `instanceof` τ, and `checkcast` τ are presented. Accesses to symbolic objects (*e.g.*, `getfield` f) operate according to the *lazy initialization* algorithm described previously. Similar to [37], we limit the choosing range to symbolic objects and arrays by introducing an additional field, CONC, which is defined for concrete objects while undefined for symbolic objects. This eliminates false alarms in the case where freshly created objects (using the `new` τ instruction during the execution) are reachable through object expansion; concretely, this only happens through assignments.

- Instruction `new` τ creates a fresh object of type τ and puts it into heap. By the definition of *new-obj*, all the fields including CONC are initialized. This guarantees that the newly created object will not be in the choosing ranges of lazy initializations.
- Instruction `getfield` f reads the f field of an object which is indexed by the heap address on the top of the stack. Semantic rules GETFIELD(1..7)-S are for `getfield`. Rules GETFIELD1-S and GETFIELD7-S are the default behavior of the `getfield` f: GETFIELD1-S is for the case that the field of the object is defined; GETFIELD7-S is for the case that the object reference is NULL. Rules GETFIELD(2..6)-S demonstrate the lazy initialization algorithm when the field is undefined. GETFIELD2-S handles the sub-case of primitive field type. A new symbol is created and the field is initialized with the fresh symbol. GETFIELD3-S lazily initializes a non-primitive field to NULL. GETFIELD4-S lazily initializes a non-primitive field by nondeterministically choosing from existing symbolic objects (with CONC undefined) from heap with compatible types. Rules GETFIELD5-S and GETFIELD6-S show the field is initialized with a new symbolic object or array respectively if the object bound is not exhausted (greater than zero).
- Instruction `putfield` τ writes a value to a field of an object. The value and heap address of the object are on the top of the stack. There are two rules for `putfield` τ: PUTFIELD1-S and PUTFIELD2-S. PUTFIELD1-S handles the normal case and PUTFIELD2-S deals with the case that the object is NULL.
- Instruction `instanceof` τ tests whether an object is a type of τ. According to the JVM specification [43], if the object is NULL, the test returns true. If the object is non-NULL, returns true if the type of the object is a subtype of τ, false otherwise. Rule INSTANCEOF1-S represents the NULL case and INSTANCEOF2-S does the non-NULL case.

  - Instruction `checkcast` τ is very similar to the instruction `instanceof`. The difference is the return: if the test passes, it does nothing; otherwise it throws a ClassCastException exception.

- Array manipulation instructions: only `anewarray` τ, `iastore`, and `iaload` are presented here. As mentioned previously, symbolic arrays present a unique challenge: fields of symbolic objects are fixed by their types but elements of symbolic arrays are not fixed because the length may be unknown; this includes arrays explicitly created with a symbolic length. To address this, we introduce another bound n on symbol $X^{m,n}$ that limits the number of distinct array elements that can be lazily initialized, that is, each symbolic array allows lazy initializations up to n kinds of distinct elements. If an array element is accessed through a symbolic index (*e.g.*, `iaload`), there are three possibilities:

  1. the index maybe out of bounds,
  2. the index is equal to one of the accessed indexes (from the *acc-idx* function), or
  3. n is decremented if the above does not hold, the number of distinct indexes accessed so far is less than the length of the array, and n is greater than zero.

  Elements of local arrays (created by `anewarray`) should have default values, but we cannot simply assign default values to all elements to a local array because the array length maybe unknown. Instead, we keep a default value for the array on its DEF field and lazily initialize an accessed index with it.

  - Instruction `anewarray` τ creates a new array with its length equals to the value on the top of the stack. Because the way we bound arrays, there are two rules for this instruction: ANEWARRAY1-S for fixed (concrete) array length and the array bound is the same as the length; ANEWARRAY2-S for symbolic array length.
  - Instruction `iastore` writes an integer value into an integer array. Rule IASTORE1-S is for the array index out of bound case. IASTORE4-S presents the case that the array is

NULL. Rule IASTORE2 is for the case that the index equals to one of accessed indexes. Rule IASTORE3 creates a new index in the array.

  - Instruction `iaload` reads the value from an index of an array. Similar to `getfield`, lazy initialization is applied when an index is undefined (Rule IALOAD3-S). The rest rules are similar to the rules for instruction `iastore`.

- Control transfer instructions: we list semantic rules for instructions `if_icmplt` and `if_acmpeq`.

  - Instruction `if_icmplt` compares the two top integral values on the stack. Since the two compared values may be symbolic and thus the ordering may not be decided, rule IF_ICMPLT-S has two end states to cover both the true and false branches. If the comparison can be determined, one branch will have inconsistent path condition, which will then be ignored.

  - Instruction `if_acmpeq` compares two object references on the top of the stack. Since Kiasan maintains a precise visible heap, the two references are either equal or not equal. Thus there are two rules for `if_acmpeq`: IF_ACMPEQ1-S for not equal case and IF_ACMPEQ2-S for the equal case.

- Instructions `assume` and `assert` instructions: the semantics for `assume` and `assert` are standard: if the top of the stack is true, `assume` and `assert` does nothing; otherwise, `assume` terminates the execution silently by making path condition FALSE, while `assert` signals an error and terminates the execution.

We use the binding, $\sigma = (g, pc, l, \omega, h, \phi)$, for all the rules. And k is used as both the object bound and the array bound.

$$\text{IADD-S}\ \frac{code(pc) = \texttt{iadd} \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_S (g, next(pc), l, Y :: \omega', h, \phi \cup \{Y = v_1 + v_2\})}$$
$$\text{where } Y = \textit{new-prim-sym}(\textsc{int}, symbols(\sigma))$$

$$\text{IF\_ICMPLT-S}\ \frac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_S (g, next(pc), l, \omega', h, \phi \cup \{v_2 \not< v_1\}) \parallel (g, pc', l, \omega', h, \phi \cup \{v_2 < v_1\})}$$

$$\text{NEW-S}\ \frac{code(pc) = \texttt{new}\ \tau \qquad i \notin \operatorname{dom} h}{\sigma \Rightarrow_S (g, next(pc), l, i :: \omega, h[i \mapsto \textit{new-obj}(symbols(\sigma), \tau)], \phi)}$$

$$\text{GETFIELD1-S}\ \frac{code(pc) = \mathtt{getfield}\ f_\tau \qquad \omega = i :: \omega' \qquad h(i)(f_\tau)\downarrow}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, h(i)(f_\tau) :: \omega', h, \phi)}$$

$$\text{GETFIELD2-S}\ \frac{code(pc) = \mathtt{getfield}\ f_\tau \qquad \omega = i :: \omega' \qquad h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{prim}}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, X :: \omega', h[i \mapsto h(i)[f \mapsto X]], \phi)}$$
$$\text{where } X = \textit{new-prim-sym}(\tau, symbols(\sigma))$$

$$\text{GETFIELD3-S}\ \frac{code(pc) = \mathtt{getfield}\ f_\tau \qquad \omega = i :: \omega' \qquad h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{non-prim}}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \textsc{null} :: \omega', h[i \mapsto h(i)[f \mapsto \textsc{null}]], \phi)}$$

$$\text{GETFIELD4-S}\ \frac{\begin{array}{c} code(pc) = \mathtt{getfield}\ f_\tau \\ \omega = i :: \omega' \qquad h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{non-prim} \qquad j \in collect(h) \qquad Z_{\tau'} = h(j) \end{array}}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]], \phi \cup \{\tau' <: \tau\})}$$

$$\text{GETFIELD5-S}\ \frac{\begin{array}{c} code(pc) = \mathtt{getfield}\ f_\tau \qquad \omega = i :: \omega' \\ h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{array} \qquad Y^{m,n} = h(i) \qquad m > 0 \qquad j \notin \operatorname{dom} h \end{array}}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau, Z(\textsc{len}) \geq 0\})}$$
$$\text{where } Z_{\tau'} = \textit{new-sarr}(symbols(\sigma), m-1, k)$$

$$\text{GETFIELD6-S}\ \frac{\begin{array}{c} code(pc) = \mathtt{getfield}\ f_\tau \qquad \omega = i :: \omega' \\ h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{record} \qquad Y^{m,n} = h(i) \qquad m > 0 \qquad j \notin \operatorname{dom} h \end{array}}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\})}$$
$$\text{where } Z_{\tau'} = \textit{new-sym}(symbols(\sigma), m-1, k)$$

$$\text{GETFIELD7-S}\ \frac{code(pc) = \mathtt{getfield}\ f_\tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

$$\text{PUTFIELD1-S}\ \frac{code(pc) = \mathtt{putfield}\ f \qquad \omega = v :: i :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h[i \mapsto h(i)[f \mapsto v]], \phi)}$$

$$\text{PUTFIELD2-S}\ \frac{code(pc) = \mathtt{putfield}\ f \qquad \omega = v :: \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

$$\text{ANEWARRAY1-S}\ \frac{code(pc) = \mathtt{anewarray}\ \tau \qquad \omega = m :: \omega' \qquad i \notin \operatorname{dom} h}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h[i \mapsto \textit{new-arr}(symbols(\sigma), \tau, m, m)], \phi)}$$

$$\text{ANEWARRAY2-S}\ \frac{code(pc) = \mathtt{anewarray}\ \tau \qquad \omega = X :: \omega' \qquad i \notin \operatorname{dom} h}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h[i \mapsto \textit{new-arr}(symbols(\sigma), \tau, X, k)], \phi \cup \{X \geq 0\}) \ ||}$$
$$\text{NegativeArraySizeException}, (g, pc, l, \omega', h, \phi \cup \{X < 0\})$$

$$\text{IASTORE1-S}\ \frac{code(pc) = \mathtt{iastore} \qquad \omega = v :: \iota :: i :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} \text{ArrayIndexOutOfBoundException}, (g, pc, l, \omega', h, \phi \cup \{\iota < 0 \vee \iota \geq h(i)(\textsc{len})\})}$$

$$\text{IASTORE2-S}\ \frac{code(pc) = \mathtt{iastore} \qquad \omega = v :: \iota :: i :: \omega' \qquad Z = h(i) \qquad \iota' \in \textit{acc-idx}(Z)}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h[i \mapsto Z[\iota' \mapsto v]], \phi \cup \{\iota = \iota'\})}$$

$$\text{IASTORE3-S}\ \frac{\begin{array}{c} code(pc) = \mathtt{iastore} \\ \omega = v :: \iota :: i :: \omega' \qquad Z^{m,n} = h(i) \qquad n > 0 \qquad I = \textit{acc-idx}(Z) \end{array}}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h[i \mapsto Z^{m,n-1}[\iota \mapsto v]], \phi \cup \{\, \iota \neq \iota' \mid \iota' \in I \,\}}$$
$$\cup \{0 \leq \iota, \iota < Z(\textsc{len}), |I| < Z(\textsc{len})\})$$

$$\text{IASTORE4-S}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = v :: \iota :: \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

$$\text{IALOAD1-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: i :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \phi \cup \{\iota < 0 \vee h(i)(\textsc{len}) \leq \iota\})}$$

$$\text{IALOAD2-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: i :: \omega' \qquad Z = h(i) \qquad \iota' \in acc\text{-}idx(Z)}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, Z(\iota') :: \omega', h, \phi \cup \{\iota = \iota'\})}$$

$$\text{IALOAD3-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: i :: \omega' \qquad Z^{m,n} = h(i) \qquad I = acc\text{-}idx(Z^{m,n})}{\begin{array}{c}\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, v :: \omega', h[i \mapsto Z^{m,n-1}[\iota \mapsto v]], \phi \cup \{\, \iota' \neq \iota \mid \iota' \in I\,\} \\ \cup \{0 \leq \iota, \iota < Z^{m,n}(\textsc{len}), |I| < Z^{m,n}(\textsc{len}), n > 0\})\end{array}}$$

$$\text{where } v = \begin{cases} Z^{m,n}(\textsc{def}) & \text{if } Z^{m,n}(\textsc{def})\downarrow \\ new\text{-}prim\text{-}sym(\text{INT}, symbols(\sigma)) & \text{if } Z^{m,n}(\textsc{def})\uparrow \end{cases}$$

$$\text{IALOAD4-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ1-S}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v_2 :: v_1 :: \omega' \qquad v_2 \neq v_1}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ2-S}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v_2 :: v_1 :: \omega' \qquad v_2 = v_1}{\sigma \Rightarrow_{\mathcal{S}} (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNULL1-S}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = i :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{IFNULL2-S}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNONNULL1-S}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = i :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNONNULL2-S}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{INSTANCEOF1-S}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, 1 :: \omega', h, \phi)}$$

$$\text{INSTANCEOF2-S}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau'} = h(i)}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, 1 :: \omega', h, \phi \cup \{\tau' <: \tau\}) \parallel (g, next(pc), l, 0 :: \omega', h, \phi \cup \{\tau' \not<: \tau\})}$$

$$\text{CHECKCAST1-S}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \textsc{null} :: \omega', h, \phi)}$$

$$\text{CHECKCAST2-S}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau'} = h(i)}{\begin{array}{c}\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, i :: \omega', h, \phi \cup \{\tau' <: \tau\}) \parallel \\ \text{ClassCastException}, (g, pc, l, i :: \omega', h, \phi \cup \{\tau' \not<: \tau\})\end{array}}$$

$$\text{ASSUME-S}\ \frac{code(pc) = \texttt{assume} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi \cup \{v = 1\})}$$

$$\text{ASSERT-S}\ \frac{code(pc) = \texttt{assert} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi \cup \{v = 1\}) \parallel \textsc{Error}, (g, pc, l, \omega', h, \phi \cup \{v = 0\})}$$

5.2.2 Operational Semantics of Symbolic Execution with Lazier Initialization

First we introduce a new semantic domain: the set of symbolic locations, **SymLocs**, to model unknown non-NULL references. We let δ ranges over symbolic locations and each $\delta_{\tau}^{m,n}$ has the same three properties as non-primitive symbols do. Clearly, we need to add the symbolic locations into domain **Values**. So we have **Values** = **Consts** ∪ **Locs** ∪ **Symbols**$_{prim}$ ∪ **SymLocs**. We use Σ_a[3] to denote the set of SELA states.

The only difference between SELA and SEL states is that the SELA states can have symbolic locations. Thus $\Sigma_a \supset \Sigma_s$.

Auxiliary Functions

We introduce some auxiliary functions to facilitate the definition of operational semantics of lazier initialization. *init-loc-heap* returns the modified heap and new constraints introduced by initializing a symbolic location to a location. *init-sym-loc* transforms a SELA state into a new SELA state by initializing a symbolic location into a location. *init-sym-loc** takes a SELA state and a symbolic location and returns a set of SELA states with the symbolic location initialized.

$$\textit{init-loc-heap} : (\mathbf{Heaps}_a \times \mathcal{P}(\mathbf{Symbols}) \times \mathbf{SymLocs} \times \mathbf{Locs}) \to (\mathbf{Heaps}_a \times \Phi))$$

$$\textit{init-sym-loc} : \Sigma_a \times \mathbf{SymLocs} \times \mathbf{Locs} \to \Sigma_a$$

$$\textit{init-sym-loc}^* : \Sigma_a \times \mathbf{SymLocs} \to \mathcal{P}(\Sigma_a).$$

The definitions are listed as follows with binding $\sigma_a = (g, pc, l, h, \phi)$:

- the *init-loc-heap* function: $\textit{init-loc-heap}(h_a, ss, \delta_{\tau}^{m,n}, i) = (h'_a, \phi')$ where
  - if $i \in \operatorname{dom} h_a$: $h'_a = \textit{sub-fun2}_1(h_a, \delta, i)$ and

$$\phi' = \{\tau' <: \tau\} \text{ where } h_a(i) = X_{\tau'}.$$

[3]Subscript a denotes that the component is a part of SELA state.

– if $i \notin \operatorname{dom} h_a$:

$$\operatorname{dom} h'_a = \operatorname{dom} h_a \cup \{i\}$$

and

$$\forall j \in \operatorname{dom} h_a . h'_a(j) = \textit{sub-fun}_1(h_a(j), \delta_\tau, i)$$

and $h'_a(i) = X_{\tau'}$ where

$$X_{\tau'} = \begin{cases} \textit{new-sarr}(ss, m, k) & \text{if } \tau \in \mathbf{Types}_{array} \\ \textit{new-sym}(ss, m, k) & \text{if } \tau \in \mathbf{Types}_{record} \end{cases}$$

and

$$\phi' = \begin{cases} \{X(\textsc{len}) \geq 0, \tau <: \tau'\} & \text{if } \tau \in \mathbf{Types}_{array} \\ \{\tau' <: \tau\} & \text{if } \tau \in \mathbf{Types}_{record} \end{cases}.$$

- *init-sym-loc* function,

$$\textit{init-sym-loc} = \lambda(\sigma_a, \delta_\tau^{m,n}, i).\{(\textit{sub-fun}_1(g, \delta, i), pc, \textit{sub-fun}_1(l, \delta, i), \textit{sub-seq}_1(\omega, \delta, i),$$
$$\#1(\textit{init-loc-heap}(h, \textit{symbols}(\sigma_a), \delta_\tau^{m,n}, i)), \#2(\textit{init-loc-heap}(h, \textit{symbols}(\sigma_a), \delta_\tau^{m,n}, i)) \cup \phi)\}$$

- *init-sym-loc** function,

$$\textit{init-sym-loc}^* = \lambda(\sigma_a, \delta_\tau^{m,n}).\{\textit{init-sym-loc}(\sigma_a, \delta_\tau^{m,n}, i) \mid i \in \textit{collect}(h)$$
$$\text{or } i \in (\mathbf{Locs} \setminus \operatorname{dom} h) \text{ if } m \geq 0\}.$$

In general, the SELA semantic rules are the same as SEL semantics rules if all the operands are not symbolic locations; otherwise, initializations of the symbolic locations in the operands will be done first. We show the SELA semantic rules for instructions `if_acmpeq` and `getfield` below. There are two notable features in the operational semantics for SELA. First, the rules are "small step". For example, there are three semantics rules for the `if_acmpeq` instruction: the two rules just initialize the operands if either operand is a symbolic location (the program counter does not change); if two operands are locations, then rule IF_ACMPEQ1-S or IF_ACMPEQ2-S will apply. Second, instead of using a symbolic location to represent all the candidates (NULL, existing objects, and a new symbolic object) for return, the `getfield` rule treats the NULL case separately, thus for a reference field access, the `getfield` will return a non-deterministic choice between NULL (rule GETFIELD3-S) and a symbolic location which denotes a non-NULL unknown reference (rule GETFIELD2-A). This is because there are usually a lot of null-ness tests in Java code and specifications; and we still want to take advantage of lazier initialization after a null-ness test. So, for `getfield`, the rules GETFIELD1,2,3,7-S stay the same in the lazier initialization and rules GETFIELD4,5,6-S are replaced by GETFIELD2-A.

Similar to the semantics rules of SEL, we use the binding $\sigma = (g, pc, l, \omega, h, \phi)$ and all the end states with path conditions that are unsatisfiable are ignored.

$$\text{IF\_ACMPEQ1-A}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \delta_\tau^{m,n} :: \delta_\tau^{m,n} :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} (g, pc', \omega', h, \phi))}$$

$$\text{IF\_ACMPEQ2-A}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \delta_\tau^{m,n} :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-loc}^*(\sigma, \delta_\tau^{m,n})}$$

$$\text{IF\_ACMPEQ3-A}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \delta_\tau^{m,n} :: v :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-loc}^*(\sigma, \delta_\tau^{m,n})}$$

$$\text{IFNULL-A}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \delta :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{IFNONNULL-A}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \delta :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} (g, pc', l, \omega', h, \phi)}$$

$$\text{GETFIELD1-A}\ \frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = \delta_\tau^{m,n} :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-loc}^*(\sigma, \delta_\tau^{m,n})}$$

$$\text{GETFIELD2-A}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \\ \omega = i :: \omega' \qquad Y^{m,n} = h(i) \qquad Y(f_\tau)\uparrow \qquad \tau \in \textbf{Types}_{non-prim} \qquad \delta \text{ is fresh} \end{array}}{\sigma \Rightarrow_{\mathcal{A}} (g, next(pc), l, \delta_\tau^{m-1,k} :: \omega', h[i \mapsto Y^{m,n}[f_\tau \mapsto \delta_\tau^{m-1,k}]], \phi)}$$

5.2.3 Operational Semantics of Symbolic Execution with Lazier# Initialization

First we introduce a new semantic domain: the set of symbolic references, **SymRefs**, to model unknown non-NULL references or NULL. We let $\hat{\delta}$ ranges over **SymRefs**. Each $\hat{\delta}_\tau^{m,n}$ is the same as $\delta_\tau^{m,n}$ except that it can be initialized to NULL. Clearly, we need to add the new domain into the domain **Values**. So we have

$$\textbf{Values} = \textbf{Consts} \cup \textbf{Locs} \cup \textbf{Symbols}_{prim} \cup \textbf{SymLocs} \cup \textbf{SymRefs}.$$

We use Σ_b[4] to denote the set of SELB states. Clearly $\Sigma_b \supset \Sigma_a$.

Auxiliary Functions

Similar to SELA, some auxiliary functions are introduced to facilitate the definition of operational semantics of SELB:

$$\begin{aligned}
&\textit{init-sym-ref} : \Sigma_b \times \textbf{SymRefs} \times (\textbf{SymLocs} \cup \{\text{NULL}\}) \to \Sigma_b \\
&\textit{init-sym-ref}^* : \Sigma_b \times \textbf{SymRefs} \to \mathcal{P}(\Sigma_b).
\end{aligned}$$

The definitions are listed as follows with binding $\sigma_b = (g, pc, l, \omega, h, \phi)$:

- *init-sym-ref* function,

$$\begin{aligned}
\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \text{NULL}) = \{(&\textit{sub-fun}_1(g, \hat{\delta}, \text{NULL}), pc, \textit{sub-fun}_1(l, \hat{\delta}, \text{NULL}), \\
&\textit{sub-seq}_1(\omega, \hat{\delta}, \text{NULL}), \textit{sub-fun2}_1(h, \hat{\delta}, \text{NULL}), \phi)\}
\end{aligned}$$

  and

$$\begin{aligned}
\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) = \{(&\textit{sub-fun}_1(g, \hat{\delta}, \delta), pc, \textit{sub-fun}_1(l, \hat{\delta}, \delta), \textit{sub-seq}_1(\omega, \hat{\delta}, \delta), \\
&\textit{sub-fun2}_1(h, \hat{\delta}, \delta), \phi)\}
\end{aligned}$$

- *init-sym-ref** function,

$$\begin{aligned}
\textit{init-sym-ref}^*(\sigma_b, \hat{\delta}) = &\{\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) \mid \delta \notin \textit{collect-sym-locs}(\sigma_b)\} \\
&\cup \{\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \text{NULL})\}.
\end{aligned}$$

In general, the semantic rules of the SELB are the same as those of the SELA if all the operands are not symbolic references; otherwise, initializations of the operands that are symbolic references will be done first. We show the SELB semantic rules for instructions `if_acmpeq` and `getfield` below. For instruction `getfield`, instead of returning a non-deterministic choice between NULL and a symbolic location, rule GETFIELD2-B just returns a fresh symbolic reference.

[4] Subscript b denotes that the component is a part of SELB state.

So, for `getfield`, the rules GETFIELD1,2,7-S and GETFIELD1-A stay the same in the lazier# initialization and rules GETFIELD3, 4,5,6-S are replaced by GETFIELD2-A.

Similar to the symbolic semantics rules, we use the binding $\sigma = (g, pc, l, \omega, h, \phi)$ and all the end states with path conditions unsatisfiable are ignored.

$$\text{IF\_ACMPEQ1-B}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\delta}^{m,n}_{\tau} :: \hat{\delta}^{m,n}_{\tau} :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} (g, pc', \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ2-B}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \hat{\delta}^{m,n}_{\tau} :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-ref}^{*}(\sigma, \hat{\delta}^{m,n}_{\tau})}$$

$$\text{IF\_ACMPEQ3-B}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\delta}^{m,n}_{\tau} :: v :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-ref}^{*}(\sigma, \hat{\delta}^{m,n}_{\tau})}$$

$$\text{GETFIELD1-B}\ \frac{code(pc) = \texttt{getfield}\ f_{\tau} \qquad \omega = \hat{\delta}^{m,n}_{\tau} :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-ref}^{*}(\sigma, \hat{\delta}^{m,n}_{\tau})}$$

$$\text{GETFIELD2-B}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_{\tau} \\ \omega = i :: \omega' \qquad Y^{m,n} = h(i) \qquad Y(f_{\tau})\uparrow \qquad \tau \in \mathbf{Types}_{non-prim} \qquad \hat{\delta} \text{ is fresh} \end{array}}{\sigma \Rightarrow_{\mathcal{B}} (g, next(pc), l, \hat{\delta}^{m-1,k}_{\tau} :: \omega', h[i \mapsto Y^{m,n}[f_{\tau} \mapsto \hat{\delta}^{m-1,k}_{\tau}]], \phi)}$$

5.2.4 Operational Semantics of JVM Bytecode Concrete Execution

To prove properties of symbolic executions in Kiasan, we need to formalize the concrete JVM bytecode execution. We introduce concrete states:

$$\sigma_c \in \Sigma_c = \mathbf{Globals} \times \mathbf{PCs} \times \mathbf{Locals} \times \mathbf{Stacks} \times \mathbf{Heaps} \times \text{Boolean}.$$

Compared to the symbolic states, concrete states have three restrictions:

1. no $X \in \mathbf{Symbols}_{prim}$ appears in concrete states;
2. no **SymTypes** appears in the concrete states;
3. for all $X_{\tau} \in \mathbf{Symbols}_{non-prim}$ which appears in concrete states, all the fields of type τ are defined and there is no bound associated with X. Furthermore, DEF and CONC are removed from the **Fields** domain.

We also need to change the definition of *new-arr* to $\textit{new-arr}_c : \mathcal{P}(\textbf{Symbols}) \times \textbf{Types} \times \mathbb{N} \rightarrow \textbf{Symbols}_{non-prim} =$

$$\lambda(ss, \tau, m).X_{\tau'}, \text{ s.t. } X \notin ss \wedge \tau' = \textit{array-type}(\tau) \wedge$$
$$\forall 0 \leq j < m.X_{\tau'}(j) = \textit{default}(\tau) \wedge X_{\tau'}(\textsc{len}) = m.$$

The concrete JVM bytecode operational semantic rules are listed below. We use the binding $\sigma = (g, pc, l, \omega, h, \textsc{True})$ for all the rules. When the last component of the end state is FALSE, the transition is ignored. Note that we do not use the wrap around semantics for integral types because it complicates the operational semantics presentation. In addition, we do not check bugs introduced by integer wrap around in symbolic executions. However, wrap around can be supported by using appropriate decision procedures that model integers using bit-vectors.

$$\text{IADD-C}\ \frac{\textit{code}(pc) = \texttt{iadd} \qquad \omega = c :: d :: \omega'}{\sigma \Rightarrow_C (g, \textit{next}(pc), l, (c+d) :: \omega', h, \textsc{True})}$$

$$\text{IF\_ICMPLT1-C}\ \frac{\textit{code}(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = d :: c :: \omega' \qquad c < d}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \textsc{True})}$$

$$\text{IF\_ICMPLT2-C}\ \frac{\textit{code}(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = d :: c :: \omega' \qquad c \not< d}{\sigma \Rightarrow_C (g, \textit{next}(pc), l, \omega', h, \textsc{True})}$$

$$\text{IF\_ACMPEQ1-C}\ \frac{\textit{code}(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = i :: j :: \omega' \qquad i \neq j}{\sigma \Rightarrow_C (g, \textit{next}(pc), l, \omega', h, \textsc{True})}$$

$$\text{IF\_ACMPEQ2-C}\ \frac{\textit{code}(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = i :: j :: \omega' \qquad i = j}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \textsc{True})}$$

$$\text{IFNULL1-C}\ \frac{\textit{code}(pc) = \texttt{ifnull}\ pc' \qquad \omega = i :: \omega'}{\sigma \Rightarrow_C (g, \textit{next}(pc), l, \omega', h, \phi)}$$

$$\text{IFNULL2-C}\ \frac{\textit{code}(pc) = \texttt{ifnull}\ pc' \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNONNULL1-C}\ \frac{\textit{code}(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = i :: \omega'}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNONNULL2-C}\ \frac{\textit{code}(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, \textit{next}(pc), l, \omega', h, \phi)}$$

$$\text{ANEWARRAY1-C}\ \frac{\textit{code}(pc) = \texttt{anewarray}\ \tau \qquad \omega = c :: \omega' \qquad c \geq 0 \qquad i \notin \text{dom}\, h}{\sigma \Rightarrow_C (g, \textit{next}(pc), l, i :: \omega', h[i \mapsto \textit{new-arr}_c(\textit{symbols}(\sigma), \tau, c)], \textsc{True})}$$

$$\text{ANEWARRAY2-C}\ \frac{\textit{code}(pc) = \texttt{anewarray}\ \tau \qquad \omega = c :: \omega' \qquad c < 0}{\sigma \Rightarrow_C \text{NegativeArraySizeException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{NEW-C}\ \frac{code(pc) = \texttt{new}\ \tau \qquad i \notin \mathrm{dom}\, h}{\sigma \Rightarrow_C (g, next(pc), l, i :: \omega, h[i \mapsto new\text{-}obj(symbols(\sigma), \tau)], \textsc{True})}$$

$$\text{IASTORE1-C}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = d :: c :: i :: \omega' \qquad c < 0 \vee c \geq h(i)(\textsc{len})}{\sigma \Rightarrow_C \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{IASTORE2-C}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = d :: c :: i :: \omega' \qquad 0 \leq c < h(i)(\textsc{len})}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h[i \mapsto h(i)[c \mapsto d]], \textsc{True})}$$

$$\text{IASTORE3-C}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = d :: c :: \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, , \textsc{True})}$$

$$\text{IALOAD1-C}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = c :: i :: \omega' \qquad c < 0 \vee c \geq h(i)(\textsc{len})}{\sigma \Rightarrow_C \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{IALOAD2-C}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = c :: \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{IALOAD3-C}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = c :: i :: \omega' \qquad 0 \leq c < h(i)(\textsc{len})}{\sigma \Rightarrow_C (g, next(pc), l, h(i)(c) :: \omega', h, \textsc{True})}$$

$$\text{GETFIELD1-C}\ \frac{code(pc) = \texttt{getfield}\ f \qquad \omega = i :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, h(i)(f) :: \omega', h, \textsc{True})}$$

$$\text{GETFIELD2-C}\ \frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{PUTFIELD1-C}\ \frac{code(pc) = \texttt{putfield}\ f \qquad \omega = v :: i :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h[i \mapsto h(i)[f \mapsto v]], \textsc{True})}$$

$$\text{PUTFIELD2-C}\ \frac{code(pc) = \texttt{putfield}\ f \qquad \omega = v :: \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{INSTANCEOF1-C}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, 1 :: \omega', h, \textsc{True})}$$

$$\text{INSTANCEOF2-C}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 <: \tau}{\sigma \Rightarrow_C (g, next(pc), l, 1 :: \omega', h, \textsc{True})}$$

$$\text{INSTANCEOF2-C}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 \not<: \tau}{\sigma \Rightarrow_C (g, next(pc), l, 0 :: \omega', h, \textsc{True})}$$

$$\text{CHECKCAST1-C}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \textsc{null} :: \omega', h, \textsc{True})}$$

$$\text{CHECKCAST2-C}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 <: \tau}{\sigma \Rightarrow_C (g, next(pc), l, i :: \omega', h, \textsc{True})}$$

$$\text{CHECKCAST2-C}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 \not<: \tau}{\sigma \Rightarrow_C \text{ClassCastException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{ASSUME1-C}\ \frac{code(pc) = \texttt{assume} \qquad \omega = 0 :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{False})}$$

$$\text{ASSUME2-C}\ \frac{code(pc) = \texttt{assume} \qquad \omega = 1 :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{True})}$$

$$\text{ASSERT1-C}\ \frac{code(pc) = \texttt{assert} \qquad \omega = 0 :: \omega'}{\sigma \Rightarrow_C \textsc{Error}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{ASSERT2-C}\ \frac{code(pc) = \texttt{assert} \qquad \omega = 1 :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{True})}$$

5.3 Proofs of Soundness and Completeness

In this section, we will prove the soundness and completeness for noncompositional symbolic executions with lazy/lazier/lazier# initialization based on the concrete execution.

5.3.1 Relative Soundness and Completeness of Symbolic Execution with Lazy Initialization

In this section, we relate the symbolic execution (non-compositional) with lazy initialization and the concrete execution under the assumption the object and array bounds are sufficient large. First we will define a concretization function γ_s [5] to relate SEL states and concrete state. Second, we will introduce binary relations between concrete and SEL states and prove simulation between concrete state-space and SEL state-space. Finally, we will prove the relative soundness and completeness of SEL regarding to concrete execution.

Definition of γ_s

Let us start with some definitions:

- the set of all environments, $Env = \left\{ E \mid E : \mathbf{Symbols}_{prim} \rightarrow \mathbf{Consts} \right\}$;
- the set of all type environments, $\Gamma = \left\{ T \mid T : \mathbf{SymTypes} \rightarrow (\mathbf{Types}_{array} \uplus \mathbf{Types}_{record}) \right\}$;
- the group of all permutations of Locations, Sym(**Locs**).

[5] Since we assume the ideal case: the object bound and array length bounds k are sufficient large, any symbol or array always has bounds greater than 0.

Then we introduce some semantic functions[6] to facilitate the definition of γ_s.

$$
\begin{aligned}
\mathcal{V}_s &: \mathbf{Values}_s \to ((Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathbf{Values}_c) \\
\mathcal{O}_s &: \mathbf{Symbols}_{non-prim} \to ((\Gamma \times Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathcal{P}(\mathbf{Symbols}_{non-prim})) \\
\mathcal{H}_s &: \mathbf{Heaps}_s \to ((\Gamma \times Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathcal{P}(\mathbf{Heaps}_c)) \\
\mathcal{ST}_s &: \Sigma_{\mathcal{S}} \to ((\Gamma \times Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathcal{P}(\Sigma_C)).
\end{aligned}
$$

Here are the definitions ($\forall T \in \Gamma, E \in Env, \rho \in \mathrm{Sym}(\mathbf{Locs})$):

- the $\mathcal{V}_s$ function:

$$\mathcal{V}_s[\![v]\!](E,\rho) = sub(sub(v,E),\rho)$$

- the $\mathcal{O}_s$ function:

$$\mathcal{O}_s[\![X_\tau]\!](T,E,\rho) = \{\, X'_{\tau'} \mid \tau' = sub(\tau,T) \wedge mapfields(X, X'_{\tau'}, E, \rho)\,\},$$

  where

$$
\begin{aligned}
mapfields(X, X'_{\tau'}, E, \rho) &\overset{\text{def}}{=} \forall \iota. X(\iota)\downarrow \implies X'(\iota) = \mathcal{V}_s[\![X(\iota)]\!](E,\rho), \text{ if } \tau' \in \mathbf{Types}_{record} \\
mapfields(X, X'_{\tau'}, E, \rho) &\overset{\text{def}}{=} X'(\textsc{len}) = \mathcal{V}_s[\![X(\textsc{len})]\!](E,\rho) \wedge \forall \iota \in acc\text{-}idx(X). \\
&\qquad X'(\mathcal{V}_s[\![\iota]\!](E,\rho)) = \mathcal{V}_s[\![X(\iota)]\!](E,\rho) \wedge (X(\textsc{def})\downarrow \implies \forall (0 \le m < X'(\textsc{len}) \\
&\qquad \wedge\, m \notin \{\, \mathcal{V}_s[\![\iota]\!](E,\rho) \mid \iota \in acc\text{-}idx(X)\,\}).X'(m) = X(\textsc{def})), \text{if } \tau' \in \mathbf{Types}_{array}
\end{aligned}
$$

- the $\mathcal{H}_s$ function [7]:

$$
\begin{aligned}
\mathcal{H}_s[\![h_s]\!](T,E,\rho) = \{\, h_c \mid\ & contains(h_c, h_s, T, E, \rho) \wedge well\text{-}typed(h_c) \\
& \wedge well\text{-}formed(h_c, h_s, T, E, \rho)\,\},
\end{aligned}
$$

[6]From this point on, we use subscript s to denote symbolic state components and domains and c for concrete state components and domains.

[7]An alternative view of functions as sets of pairs may be taken.

where *contains*(h_c, h_s, T, E, ρ) if and only if

$$\forall (i, X) \in h_s. \exists Y \in \mathcal{O}_s[\![X]\!](T, E, \rho).(\rho(i), Y) \in h_c.$$

well-typed(h_c) if and only if for each non-primitive symbol in h_c must have all its fields mapped to values of their types. More specifically, each primitive field is mapped to a constant of its type; each reference type field is mapped to either NULL or a location in h_c which is mapped a non-primitive symbol of a compatible type.

well-formed(h_c, h_s, T, E, ρ) if and only if for each entry (i, X_c) in h_c, X_c is well-formed, that is,

1. if (i, X_c) is mapped from (j, X_s) in h_s ($i = \rho(j)$ and $X_c \in \mathcal{O}_s[\![X_s]\!](T, E, \rho)$), and if any field f of X_s is undefined and non-primitive, $X_c(f)$ has to be one of following values:
   - NULL
   - i' where $i' \notin \rho(\mathrm{dom}\, h_s)$.
   - i'' where $i'' \in \rho(\mathrm{dom}\, h_s)$ and $h_s(\rho^{-1}(i''))(\textsc{conc}) \uparrow$.
2. if (i, X_c) is not mapped from any entry in h_s ($i \notin \rho(\mathrm{dom}\, h_s)$), all the fields of X_c are treated as the ones with corresponding undefined fields in h_s.

- the $\mathcal{ST}_s$ function:

$$\mathcal{ST}_s[\![(g, pc, l, \omega, h, \phi)]\!](T, E, \rho) = \{\, (\textit{sub-fun}(\textit{sub-fun}(g, E), \rho), pc, \\ \textit{sub-fun}(\textit{sub-fun}(l, E), \rho), \textit{sub-seq}(\textit{sub-seq}(\omega, E), \rho), h', \textsc{True}) \mid h' \in \mathcal{H}_s[\![h]\!](T, E, \rho) \,\}.$$

Finally, the definition of $\gamma_s : \Sigma_s \rightarrow \mathcal{P}(\Sigma_c)$ is

$$\gamma_s(\sigma_s) = \bigcup_{\forall E, T \vDash \phi, \forall \rho} \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho).$$

Concrete and Symbolic Kripke Structures

Given a method m, we have a set of global variables G and local variables L (ordered from $0..n$). We use Kripke structures [8] $\mathcal{C} = (\Sigma_{\mathcal{C}}, I_{\mathcal{C}}, \longrightarrow_{\mathcal{C}}, L_{\mathcal{C}})$ and $\mathcal{S} = (\Sigma_{\mathcal{S}}, I_{\mathcal{S}}, \longrightarrow_{\mathcal{S}}, L_{\mathcal{S}})$ to model the state-spaces from the concrete and the SEL, respectively. Each component is defined as following

- states,

$$\Sigma_{\mathcal{C}} = \Sigma_c \cup (\textsc{Exception} \times \Sigma_c) \cup (\textsc{Error} \times \Sigma_c).$$

$$\Sigma_{\mathcal{S}} = \Sigma_s \cup (\textsc{Exception} \times \Sigma_s) \cup (\textsc{Error} \times \Sigma_s).$$

  Furthermore, we require that all the $\Sigma_{\mathcal{C}}$ and $\Sigma_{\mathcal{S}}$ are well typed according to the signature of m.

- initial states, according to JVM specification [43], the initial states have empty operand stacks and all the arguments are stored in local. So

$$I_{\mathcal{C}} = \{\, (g_c, pc_{init}, l_c, nil, h_c, \textsc{True}) \mid \mathrm{dom}(g_c) = G \wedge \mathrm{dom}(l_c) = L \,\},$$

  where pc_{init} is the start program counter of the method.

$$I_{\mathcal{S}} = \{\, (g_s, pc_{init}, l_s, nil, h_s, \{\textsc{True}\}) \mid \mathrm{dom}(g_s) = G \wedge \mathrm{dom}(l_s) = L \,\}$$

  and each local and global is initialized as follows: if its type is primitive, a symbolic primitive symbolic is created; otherwise, it is nondeterministically initialized as a symbolic object with all its fields undefined or NULL with all the possible aliasings.

- transition relations,

$$c_1 \longrightarrow_{\mathcal{C}} c_2 \iff c_1 \Rightarrow_{\mathcal{C}} c_2 \wedge \text{ last component of } c_2 \text{ is } \textsc{True}.$$

$$s_1 \longrightarrow_{\mathcal{S}} s_2 \iff s_1 \Rightarrow_{\mathcal{S}} s_2 \wedge \text{ the path condition of } s_2 \text{ is satisfiable.}$$

- labels, we will not use this component. So let them undefined.

[8] Appendix A presents definitions of Kripke structures and simulations on Kripke structures adapted from [58] for a quick reference.

Function γ_s is trivially extended to $\gamma_s^* : \Sigma_{\mathcal{S}} \to \mathcal{P}(\Sigma_C)$ as

$$\gamma_s^*(s) = \begin{cases} \gamma_s(\sigma_s), & \text{if } s = \sigma_s \text{ for some } \sigma_s \in \Sigma_s; \\ \{\, (\textsc{Exception}, \sigma_c) \mid \sigma_c \in \gamma_s(\sigma_s) \,\}, & \text{if } s = (\textsc{Exception}, \sigma_s) \text{ for some } \sigma_s \in \Sigma_s; \\ \{\, (\textsc{Error}, \sigma_c) \mid \sigma_c \in \gamma_s(\sigma_s) \,\}, & \text{if } s = (\textsc{Error}, \sigma_s) \text{ for some } \sigma_s \in \Sigma_s. \end{cases}$$

Simulation Relations

To show the relationship between C and $\mathcal{S}$, we define a relation.

Definition 1. *$\mathcal{R} \subseteq \Sigma_C \times \Sigma_{\mathcal{S}}$, as follows: $c \;\mathcal{R}\; s \iff c \in \gamma_s^*(s)$.*

For any σ_s with path condition (ϕ) satisfiable, there exists one σ_c such that $\sigma_c \;\mathcal{R}\; \sigma_s$ since there exist some E and T which satisfy ϕ.

Clearly, for all $c_0 \in I_C$, there exists $s_0 \in I_{\mathcal{S}}$ such that $c_0 \;\mathcal{R}\; s_0$.

Proposition 1. *$C \lhd_{\mathcal{R}} \mathcal{S}$.*

Proof. It is sufficient to show that for all $\sigma_c \in \Sigma_C, \sigma_s \in \Sigma_{\mathcal{S}}$ if $\sigma_c \longrightarrow_C \sigma_c'$ and $\sigma_c \;\mathcal{R}\; \sigma_s$ then there exists $\sigma_s' \in \mathcal{S}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma_s'$ and $\sigma_c' \;\mathcal{R}\; \sigma_s'$.

We will proceed with the rule induction on $\longrightarrow_C$.

- Rule IADD-C: Let $\sigma_c = (g_c, pc, l_c, d :: c :: \omega_c, h_c, \textsc{True})$, then $\sigma_c' = (g_c, \mathit{next}(pc), l_c, (c + d) :: \omega_c, h_c, \textsc{True})$. Suppose $\sigma_c \;\mathcal{R}\; \sigma_s$. We need to show that there exists $\sigma_s' \in \Sigma_{\mathcal{S}}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma_s'$ and $\sigma_c' \;\mathcal{R}\; \sigma_s'$. Since $\sigma_c \;\mathcal{R}\; \sigma_s$, we have $\sigma_c \in \gamma_s(\sigma_s)$. The symbolic state σ_s must have the form of $(g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\mathcal{V}_s[\![v_1]\!](E,\rho) = c$, $\mathcal{V}_s[\![v_2]\!](E,\rho) = d$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g_c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l_c$, $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Using the rule IADD-S, we get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma_s'$ with $\sigma_s' = (g_s, \mathit{next}(pc), l_s, Y :: \omega_s, h_s, \phi \cup \{Y = v_1 + v_2\})$ where Y is fresh. We only need to show $\sigma_c' \in \gamma_s(\sigma_s')$, that is, to find T', E', ρ' such that $\sigma_c' \in \mathcal{ST}_s[\![\sigma_s']\!](T', E', \rho')$. We claim that $T' = T$, $E' = E[Y \mapsto c + d]$, and $\rho' = \rho$ are the right choice. Since Y is fresh, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E'), \rho') = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g_c, \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E'), \rho') = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l_c, \mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E'), \rho') = \mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) =$

ω_c, and $h_c \in \mathcal{H}_s[\![h_s]\!](T', E', \rho') = \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Furthermore, since $\mathcal{V}_s[\![Y]\!](E', \rho) = c + d = \mathcal{V}_s[\![v_1]\!](E, \rho) + \mathcal{V}_s[\![v_2]\!](E, \rho)$, we get $T, E' \vDash (\phi \cup \{Y = v_1 + v_2\})$. Therefore, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T, E', \rho) \subseteq \gamma_s(\sigma'_s)$.

- Rule IF_ICMPLT2-C: Let $\sigma_c = (g_c, pc, l_c, d :: c :: \omega_c, h_c, \textsc{True})$, then $c \geq d$ and $\sigma'_c = (g_c, next(pc), l_c, \omega_c, h_c, \textsc{True})$. Suppose $\sigma_c \mathcal{R} \sigma_s$. We need to show that there exists $\sigma'_s \in \Sigma_{\mathcal{S}}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ and $\sigma'_c \mathcal{R} \sigma'_s$. Since $\sigma_c \mathcal{R} \sigma_s$, we have $\sigma_c \in \gamma_s(\sigma_s)$. The symbolic state σ_s must have the form of $(g_s, pc, l_s, v_2 :: v_1 :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\mathcal{V}_s[\![v_1]\!](E, \rho) = c$, $\mathcal{V}_s[\![v_2]\!](E, \rho) = d$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Using the IF_ICMPLT-S, we get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ with $\sigma'_s = (g_s, next(pc), l_s, \omega_s, h_s, \phi \cup \{v_1 \geq v_2\})$ (the first end state). We only need to show $\sigma'_c \in \gamma_s(\sigma'_s)$. Since $\mathcal{V}_s[\![v_1]\!](E, \rho) = c$, $\mathcal{V}_s[\![v_2]\!](E, \rho) = d$, and $c \geq d$, we get $T, E \vDash \phi \cup \{v_1 \geq v_2\}$. Therefore, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma'_s)$.

- Rule ANEWARRAY1-C: Suppose $\sigma_c = (g_c, pc, l_c, c :: \omega_c, h_c, \textsc{True})$. Then $c \geq 0$ and $\sigma'_c = (g_c, next(pc), l_c, \omega_c, h'_c, \textsc{True})$ where i is fresh and $h'_c = h_c[i \mapsto \textit{new-arr}_c(\textit{symbols}(\sigma_c), \tau, c)]$. Suppose $\sigma_c \mathcal{R} \sigma_s$. We need to show that exists $\sigma'_s \in \Sigma_{\mathcal{S}}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ and $\sigma'_c \mathcal{R} \sigma'_s$. Since $\sigma_c \mathcal{R} \sigma_s$, we have $\sigma_c \in \gamma_s(\sigma_s)$. The symbolic state σ_s has the form of $(g_s, pc, l_s, v :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\mathcal{V}_s[\![v]\!](E, \rho) = c$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Using the ANEWARRAY2-S rule, we get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ with $\sigma'_s = (g_s, next(pc), l_s, \omega_s, h'_s, \phi \cup \{v \geq 0\})$ where $h'_s = h_s[j \mapsto \textit{new-arr}(\textit{symbols}(\sigma_s), \tau, X, k)]$ and j is fresh (the first end state). We only need to show $\sigma'_c \in \gamma_s(\sigma'_s)$. Define $\rho' = \rho[j \mapsto i][\rho^{-1}(i) \mapsto \rho(j)]$. It is clear that $\rho' \in S$ and for location $i' \notin \{j, \rho^{-1}(i)\}$, $\rho'(i') = \rho(i')$. Since i is fresh in σ_c and $\sigma_c \mathcal{R} \sigma_s$, $\rho^{-1}(i)$ must be fresh in σ_s (not in $\operatorname{dom} h_s$) too. Thus we get $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho') = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho') = l_c$, and $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho') = \omega_c$. From $c \geq 0$, $\mathcal{V}_s[\![v]\!](E, \rho') \geq 0$, that is, $T, E \vDash \phi \cup \{v \geq 0\}$. It remains to show $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho')$. Clearly $\textit{well-typed}(h'_c)$ because i is fresh in h_c. Then we show that $\textit{contains}(h'_c, h'_s, T, E, \rho')$. For any entry $(i', X') \in h_s$, since j and $\rho^{-1}(i)$ are fresh in h_s, we get

$\mathcal{O}_s[\![X']\!](T,E,\rho') = \mathcal{O}_s[\![X']\!](T,E,\rho)$. Furthermore, since $\textit{new-arr}_c(\textit{symbols}(\sigma_c),\tau,c) \in \mathcal{O}_s[\![\textit{new-arr}(\textit{symbols}(\sigma_c),\tau,X,k)]\!](T,E,\rho')$, we can get $\textit{contains}(h'_c,h'_s,T,E,\rho')$. Next we need to show $\textit{well-formed}(h'_c,h_s,T,E,\rho')$. Since $\textit{new-arr}(\textit{symbols}(\sigma_s),\tau,X,k)(\text{CONC})\downarrow$, symbol $\textit{new-arr}_c(\textit{symbols}(\sigma_c),\tau,c)$ of entry $(i,\textit{new-arr}_c(\textit{symbols}(\sigma_c),\tau,c))$ in h'_c is well-formed under E and ρ'. For any symbol Y in the range of h_c, if Y has a reference field f whose corresponding field is not defined in h_s, by the $\textit{well-formed}(h_c,h_s,T,E,\rho)$, f can not be any location that points to concrete object in h_c. But h'_c has only one extra concrete object at i than h_c and i is fresh in h_c. Therefore, f can not point to i, that is, symbol $\textit{new-arr}_c(\textit{symbols}(\sigma_c),\tau,c)$ is well-formed. We get $\textit{well-formed}(h'_c,h'_s,T,E,\rho')$. Thus $h'_c \in \mathcal{H}_s[\![h'_s]\!](T,E,\rho')$. Finally, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T,E,\rho') \subseteq \gamma_s(\sigma'_s)$.

- Rule GETFIELD1-C: Suppose $\sigma_c = (g_c, pc, l_c, i :: \omega_c, h_c)$, then $\sigma'_c = (g_c, \textit{next}(pc), l_c, v :: \omega_c, h_c)$ where $X = h_c(i), v = X(f)$. Let τ_v be the real type of symbol $h_c(v)$. Suppose $\sigma_c\ \mathcal{R}\ \sigma_s$. We need to show that exists $\sigma'_s \in \Sigma_S$ such that $\sigma_s \longrightarrow_S \sigma'_s$ and $\sigma'_c\ \mathcal{R}\ \sigma'_s$. Since $\sigma_c\ \mathcal{R}\ \sigma_s$, we have $\sigma_c \in \gamma_s(\sigma_s)$. The symbolic state σ_s must have the form of $(g_s, pc, l_s, i' :: \omega_s, h_s, \phi)$ for some T,E,ρ with $T,E \vDash \phi$, $\rho(i') = i$, $\textit{sub-fun}(\textit{sub-fun}(g_s,E),\rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s,E),\rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s,E),\rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T,E,\rho)$. WLOG, assume that the type of f, τ, is a record type and f is not in the domain of $h_s(i')$. We will proceed with a case analysis according to the value of v by $\textit{well-formed}(h_c,h_s,T,E,\rho)$:
  - case $v = \text{NULL}$. We will apply the GETFIELD3-S rule and get $\sigma'_s = (g, \textit{next}(pc), l, \text{NULL} :: \omega, h'_s, \phi)$, where $h'_s = h_s[i \mapsto h_s(i)[f \mapsto \text{NULL}]]$. It suffices to show $\textit{contains}(h_c,h'_s,T,E,\rho)$ and $\textit{well-formed}(h_c,h'_s,T,E,\rho)$. Since $\rho(i') = i$ and $\sigma_c\ \mathcal{R}\ \sigma_s$, $X \in \mathcal{O}_s[\![Y]\!](T,E,\rho)$. Furthermore, Since $h_c \in \mathcal{H}_s[\![h_s]\!](T,E,\rho)$ and $X \in \mathcal{O}_s[\![h_s(i)[f_\tau \mapsto \text{NULL}]]\!](T,E,\rho)$ by $X(f) = \text{NULL} = h_s(i)(f)$, we get $\textit{contains}(h_c,h'_s,T,E,\rho)$ and $\textit{well-formed}(h_c,h'_s,T,E,\rho)$ hold. We get $h_c \in \mathcal{H}_s[\![h'_s]\!](T,E,\rho)$. Then $\sigma'_c \in \gamma_s(\sigma'_s)$.
  - case $v \in \rho(\operatorname{dom} h_s) \wedge h_s(\rho^{-1}(v))(\text{CONC})\uparrow$. We will apply the rule GETFIELD4-S and get $\sigma'_s = (g, \textit{next}(pc), l, v' :: \omega, h'_s, \phi \cup \{\tau' <: \tau\})$ where $h'_s = h_s[i \mapsto h_s(i)[f \mapsto j]]$

and $Z_{\tau'} = h_s(v')$. We also have $v' = \rho^{-1}(v)$ ($v' \in \mathit{collect}(h_s)$ because $v \in \rho(\mathrm{dom}\, h_s)$ and $h_s(\rho^{-1}(v))(\textsc{conc}) \uparrow$). Since *well-typed*$(h_c)$, the type of $h_c(v)$, τ_v, is a subtype of τ. Furthermore, since $h_c(v) \in \mathcal{O}_s[\![Z_{\tau'}]\!](T, E, \rho)$, we arrive at $T \vDash \tau' <: \tau$. Thus $T, E \vDash \phi \cup \{\tau' <: \tau\}$. The rest of the proof is similar to the NULL case.

– case $v \in \mathbf{Locs} \wedge v \notin \rho(\mathrm{dom}\, h_s)$. We will apply the rule GETFIELD6-S (because we assume that bound k is sufficient large and $m > 0$) and get $\sigma'_s = (g, \mathit{next}(pc), l, v :: \omega', h'_s, \phi \cup \{\tau' <: \tau\})$ where $h'_s = h_s[i \mapsto h_s(i)[f \mapsto j]]$ and $Z_{\tau'} = \mathit{new\text{-}sym}(\mathit{symbols}(\sigma), m - 1, k)$. Define $\rho' = \rho[j \mapsto v]$ and $T' = T[\tau' \mapsto \tau_v]$. Since *well-typed*$(h_c)$, we get $\tau_v <: \tau$. Furthermore, since $\rho' = \rho[j \mapsto v]$ and $T' = T[\tau' \mapsto \tau_v]$, $T' \vDash \tau' <: \tau$. Thus $T', E \vDash \phi \cup \{\tau' <: \tau\}$. Since j is fresh in h_s, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho') = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho)$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho') = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho)$, and $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho') = \mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho)$ hold. It remains to show $\mathit{contains}(h_c, h'_s, T, E, \rho)$ and $\mathit{well\text{-}formed}(h_c, h'_s, T, E, \rho)$. Since $X \in \mathcal{O}_s[\![Y^{m,n}[f_\tau \mapsto v']]\!](T', E, \rho')$ and $h_c(v) \in \mathcal{O}_s[\![Z_{\tau'}]\!](T', E, \rho')$, $\mathit{contains}(h_c, h'_s, T', E, \rho)$ holds. Since $v \notin \rho(\mathrm{dom}\, h_s)$, $h_c(v)$ is well-formed. Since the new symbol $Z_{\tau'}$ in h'_s has $\textsc{conc}$ field undefined, the rest of symbols in h_c are well-formed. Thus we get $\mathit{well\text{-}formed}(h_c, h'_s, T, E, \rho)$ and further, $h_c \in \mathcal{H}_s[\![h'_s]\!](T', E, \rho')$. Therefore, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T', E, \rho') \subseteq \gamma_s(\sigma'_s)$.

□

Definition 2. $\mathcal{R}_\bullet \subseteq \Sigma_{\mathcal{S}} \times \mathcal{P}(\Sigma_C)$, *as* $\sigma_s \,\mathcal{R}_\bullet\, S_c \iff \gamma^*_s(\sigma_s) = S_c$.

The relation $\mathcal{R}_\bullet$ is left total by definition. Also it is clear that for any $\sigma_s \,\mathcal{R}_\bullet\, S_c$, S_c is not empty, if and only if the path condition ϕ of σ_s is satisfiable. Furthermore, for any $\sigma_s \in I_{\mathcal{S}}$ and $\sigma_s \,\mathcal{R}_\bullet\, S_c$, it is clear that $S_c \subseteq I_C$ by the definition of γ_s function.

Proposition 2. $\mathcal{S} \lhd_{\mathcal{R}_\bullet} \mathcal{P}(C)$.

Proof. It is sufficient to show that for all $\sigma_s, \sigma'_s \in \Sigma_{\mathcal{S}}, S_c, S'_c \in \mathcal{P}(\Sigma_C)$, if $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$, $\sigma_s \,\mathcal{R}_\bullet\, S_c$, and $\sigma'_s \,\mathcal{R}_\bullet\, S'_c$ then $S_c \overset{\bullet}{\longrightarrow}_C S'_c$.

We will prove by rule induction on symbolic operational semantics transitions, $\longrightarrow_{\mathcal{S}}$.

- Rule IADD-S: $\sigma_s = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$. Then $\sigma'_s = (g_s, next(pc), l_s, Z :: \omega_s, h_s, \phi \cup \{Z = v_1 + v_2\})$ where Z is fresh. Suppose $\sigma_s \ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s \ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, next(pc), l'_c, c :: \omega'_c, h'_c, \text{TRUE})$ with some T, E, ρ such that $T, E \vDash \phi \cup \{Z = v_1 + v_2\}$, $\mathcal{V}_s[\![Z]\!](E, \rho) = c$, $sub\text{-}fun(sub\text{-}fun(g_s, E), \rho) = g'_c, sub\text{-}fun(sub\text{-}fun(l_s, E), \rho) = l'_c, sub\text{-}seq(sub\text{-}seq(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Take $\sigma_c = (g'_c, pc, l'_c, E(X) :: E(Y) :: \omega'_c, h'_c, \text{TRUE})$. Clearly $\sigma_c \longrightarrow_C \sigma'_c$. We only need to show that $\sigma_c \in \gamma_s(\sigma_s)$. Since Z is fresh, $T, E \vDash \phi$. Thus $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- Rule IF_ICMPLT-S: $\sigma_s = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$ and $\sigma'_s = (g_s, next(pc), l_s, \omega_s, h_s, \phi \cup \{v_2 \geq v_1\})$. (we only consider one end state, the other end state is symmetric.) Suppose $\sigma_s \ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s \ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, next(pc), l'_c, \omega'_c, h'_c, \text{TRUE})$ with some T, E, ρ such that $T, E \vDash \phi \cup \{v_2 \geq v_1\}$, $sub\text{-}fun(sub\text{-}fun(g_s, E), \rho) = g'_c, sub\text{-}fun(sub\text{-}fun(l_s, E), \rho) = l'_c, sub\text{-}seq(sub\text{-}seq(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Take $\sigma_c = (g'_c, pc, l'_c, \mathcal{V}_s[\![v_1]\!](E, \rho) :: \mathcal{V}_s[\![v_2]\!](E, \rho) :: \omega'_c, h'_c)$. Clearly $\sigma_c \longrightarrow_C \sigma'_c$. We conclude that $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- Rule ANEWARRARY2-S: Suppose $\sigma_s = (g_s, pc, l_s, X :: \omega_s, h_s, \phi)$. We only consider that non-exceptional end state here. Then $\sigma'_s = (g_s, next(pc), l_s, i :: \omega_s, h_s[i \mapsto new\text{-}arr(symbols(\sigma_s), \tau, X, k)], \phi \cup \{X \geq 0\})$ where i is fresh. Suppose $\sigma_s \ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s \ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, next(pc), l'_c, j :: \omega'_c, h'_c, \text{TRUE})$ with some T, E, ρ such that $T, E \vDash \phi \cup \{X \geq 0\}$, $\rho(i) = j$, $sub\text{-}fun(sub\text{-}fun(g_s, E), \rho) = g'_c, sub\text{-}fun(sub\text{-}fun(l_s, E), \rho) = l'_c, sub\text{-}seq(sub\text{-}seq(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h_s[i \mapsto new\text{-}arr(symbols(\sigma_s), \tau, X, k)]]\!](T, E, \rho)$. We need to find a $\sigma_c \in \Sigma_C$ such that $\sigma_c \in \gamma_s(\sigma_s)$ and $\sigma_c \longrightarrow_C \sigma'_c$. We claim that $\sigma_c = (g'_c, pc, l'_c, E(\alpha) :: \omega'_c, h_c, \text{TRUE})$ where $h_c = h'_c \setminus (j, h'_c(j))$

satisfies the above two conditions. Since $T, E \vDash \phi \cup \{X \geq 0\}$, $T, E \vDash \phi$. To show $\sigma_c \in \gamma_s(\sigma_s)$, it suffices to show that $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Since $\textit{new-arr}(\textit{symbols}(\sigma_s), \tau, X, k)$ will return a symbol with CONC field defined and $h'_c \in \mathcal{H}_s[\![h_s[i \mapsto \textit{new-arr}(\textit{symbols}(\sigma_s), \tau, X, k)]]\!](T, E, \rho)$, symbols in h'_c such that their corresponding symbols in $h_s[i \mapsto \textit{new-arr}(\textit{symbols}(\sigma_s), \tau, X, k)]$ have CONC fields not defined or do not have corresponding symbols can not contains j ($\rho(i)$). Furthermore, since i is fresh in h_s, h_c does not have any symbol such that j is in its range. Therefore, $\textit{well-typed}(h_c)$, $\textit{contains}(h_c, h_s, T, E, \rho)$, and $\textit{well-formed}(h_c, h_s, T, E, \rho)$. We get $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$. Clearly $\sigma_c \longrightarrow_C \sigma'_c$.

- Rule GETFIELD3-S: Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then f is not defined in $h_s(i)$ and $\sigma'_s = (g_s, \textit{next}(pc), l_s, \text{NULL} :: \omega_s, h'_s, \phi)$ where $h'_s = h_s[i \mapsto h_s(i)[f_\tau \mapsto \text{NULL}]]$. Suppose $\sigma_s\ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s\ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \textit{next}(pc), l'_c, \text{NULL} :: \omega'_c, h'_c, \text{TRUE})$ with some T, E, ρ such that $T, E \vDash \phi$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g'_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l'_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$. We need to find a σ_c such that $\sigma_c \longrightarrow_C \sigma'_c$ and $\sigma_c \in \gamma_s(\sigma_s)$. Take $\sigma_c = (g'_c, pc, l'_c, \rho(i) :: \omega'_c, h'_c, \text{TRUE})$. From $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, it is clear that $\sigma_c \longrightarrow_C \sigma'_c$. Then it suffices to show $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Since $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, $\textit{well-typed}(h'_c)$, $\textit{contains}(h'_c, h_s, T, E, \rho)$, and $\textit{well-formed}(h'_c, h_s, T, E, \rho)$ hold. Finally, $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- Rule GETFIELD6-S: Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then f is not defined in $h_s(i)$ and $\sigma'_s = (g_s, \textit{next}(pc), l_s, j :: \omega_s, h'_s, \phi')$ where $h_s(i) = Y^{m,n}$, $h'_s = h_s[i \mapsto Y^{m,n}[f_\tau \mapsto j]][j \mapsto Z_{\tau'}]$, $\phi' = \phi \cup \{\tau' <: \tau\}$ where $Z_{\tau'} = \textit{new-sym}(\textit{symbols}(\sigma_s), m - 1, k)$ and $j \notin \operatorname{dom} h_s$. Suppose $\sigma_s\ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s\ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \textit{next}(pc), l'_c, v' :: \omega'_c, h'_c, \text{TRUE})$ with some T, E, ρ such that $T, E \vDash \phi'$, $\mathcal{V}_s[\![v]\!](E, \rho) = v'$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g'_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l'_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$. We need to find a σ_c such that

$\sigma_c \longrightarrow_C \sigma'_c$ and $\sigma_c \in \gamma_s(\sigma_s)$. Define $\rho' = \rho[j \mapsto v']$ and $\sigma_c = (g'_c, pc, l'_c, \rho'(i) :: \omega'_c, h'_c, \textsc{True})$. From $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, it is clear that $\sigma_c \longrightarrow_C \sigma'_c$. Since $T, E \vDash \phi \cup \{\tau' <: \tau\}$, $T, E \vDash \phi$. Then it suffices to show $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho')$. Since $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, *well-typed*$(h'_c)$ and *contains*(h'_c, h_s, T, E, ρ) hold. Since $h'_s(j) = Z$ and $\textsc{conc} \notin \operatorname{dom} Z$, *well-formed*$(h'_c, h_s, T, E, \rho)$ hold. Finally, $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho') \subseteq \gamma_s(\sigma_s)$.

□

Relative Soundness and Completeness

The soundness means that if there is an error in the concrete execution, then the symbolic execution will be able to find it. And the completeness means that if symbolic execution finds an error, it is a real error. We use a theorem prover to decide the satisfiability of path conditions. But in general, theorem provers are neither sound nor complete for the first order logic with integer and float arithmetics. But in this section, we proceed to show the symbolic execution is sound and complete with assumption that the underlying theorem prover is sound and complete. This is why we called it "Relative Soundness and Completeness".

Proposition 3 (Soundness)**.** *Given any concrete trace $c_1 \longrightarrow_C c_2 \longrightarrow_C \cdots \longrightarrow_C c_n$ with $c_1 \in I_C$, there is a corresponding symbolic trace $s_1 \longrightarrow_S s_2 \longrightarrow_S \cdots \longrightarrow_S s_n$ with $s_1 \in I_S$ such that $c_k \mathcal{R} s_k$ for all $1 \leq k \leq n$.*

Proof. We get s_1 by the simulation relation between C and $\mathcal{S}$. Then we proceed by mathematical induction on n using Proposition 1. □

Proposition 4 (Completeness)**.** *Given any symbolic trace $s_1 \longrightarrow_S s_2 \longrightarrow_S \cdots \longrightarrow_S s_n$ with $s_1 \in I_S$, there is a corresponding concrete trace $c_1 \longrightarrow_C c_2 \longrightarrow_C \cdots \longrightarrow_C c_n$ such that $c_k \mathcal{R} s_k$ for all $1 \leq k \leq n$ and $c_1 \in I_C$.*

Proof. Since the ϕ of s_1 is not false, $C_1 = \gamma^*_s(s_1) \neq \emptyset$. Then we show there exists a trace in $\mathcal{P}(C)$, $C_1 \overset{\bullet}{\longrightarrow}_C C_2 \overset{\bullet}{\longrightarrow}_C \cdots \overset{\bullet}{\longrightarrow}_C C_n$ such that $s_k \mathcal{R}_\bullet C_k$ for all $1 \leq k \leq n$ by mathematical induction on n using Proposition 2. Since the ϕ of s_n is satisfiable, then $C_n \neq \emptyset$. Pick any $c_n \in C_n$ and use the definition of $\overset{\bullet}{\longrightarrow}_C$, we get the corresponding concrete trace $c_1 \longrightarrow_C c_2 \longrightarrow_C \cdots \longrightarrow_C c_n$. □

5.3.2 Relative Soundness and Completeness of Symbolic Execution with Lazier Initialization

Following the outline of Section 5.3.1, we relate the SELA in Section 5.2.2 and SEL in Section 5.2.1. First, we define a function γ_a which takes a SELA state and returns all the SEL states that have the same shape and only symbolic locations are initialized to concrete locations. Then we introduce binary relations between SEL states (power) and SELA states. Finally, we will prove the relative soundness and completeness of SELA with regard to SEL intraprocedurely.

Definition of γ_a

Let us first introduce a definition: The set of all symbolic variable environments

$$\Pi = \{ F \mid F : \textbf{SymLocs} \to \textbf{Locs} \}. \tag{5.1}$$

Then we define some semantic functions with subscript a denoting SELA domains/components:

$$\mathcal{H}_a : (\textbf{Heaps}_a \times \Phi) \to (\mathcal{P}(\textbf{Symbols}) \times \mathcal{P}(\textbf{SymLocs}) \times \Pi) \to \mathcal{P}(\textbf{Heaps}_s \times \Phi))$$
$$\mathcal{ST}_a : \Sigma_a \to \Pi \to \mathcal{P}(\Sigma_s).$$

The definitions [9] are listed as follows ($\forall F \in \Pi.$).

- the $\mathcal{H}_a$ function:

$$\mathcal{H}_a[\![(h_a, \phi)]\!](ss, \Delta, F) = \{(h_s, \phi') \mid \textit{well-mapped}(\Delta, h_a, F) \wedge \textit{heap}(ss, \Delta, h_a, h_s, F)$$
$$\wedge\, pc(\phi', \phi, h_s, F) \wedge \phi' \text{ is satisfiable}\},$$

  where $\textit{well-mapped} : \mathcal{P}(\textbf{SymLocs}) \times \textbf{Heaps}_a \times \Pi \to \textsc{Boolean}$ with $\textit{well-mapped}(\Delta, h_a, F)$ if and only if

$$\forall \delta \in \Delta.(h_a(F(\delta)) \uparrow \vee h_a(F(\delta))(\textsc{conc}) \uparrow);$$

[9] Subscript a is frequently used to indicate a component in the SELA states.

$heap : \mathcal{P}(\textbf{Symbols}) \times \mathcal{P}(\textbf{SymLocs}) \times \textbf{Heaps}_a \times \textbf{Heaps}_s \times \Pi \to \textsc{Boolean}$ with $heap(ss, \Delta, h_a, h_s, F)$ if and only if

$$\operatorname{dom} h_s = \operatorname{dom} h_a \cup F(\Delta) \wedge \forall i \in \operatorname{dom} h_a . h_s(i) = \textit{sub-fun}(h_a(i), F)$$
$$\wedge\ \forall i \in (\operatorname{dom} h_s - \operatorname{dom} h_a). h_s(i) = X_\tau,$$

where $X_\tau = \begin{cases} \textit{new-sarr}(ss \cup h_s(\textbf{Locs} - \{i\}), k, k), & \text{if} \exists \delta_{\tau''} \in F^{-1}(i) \text{ such that } \tau'' \in \textbf{Types}_{array} \\ \textit{new-sym}(ss \cup h_s(\textbf{Locs} - \{i\}), k, k) & \text{otherwise;} \end{cases}$

$pc : \Phi \times \Phi \times \textbf{Heaps}_s \times \Pi \times \mathcal{P}(\textbf{SymLocs}) \to \textsc{Boolean}$ with $pc(\phi', \phi, h_s, F, \Delta)$ if and only if ϕ' is the least set of predicates that satisfies following condition:

$$\phi \subseteq \phi' \wedge \forall \delta_\tau \in \Delta . \tau' <: \tau \in \phi' \wedge X(\textsc{len}) \geq 0 \in \phi' \text{ if } \tau \in \textbf{Types}_{array} \text{ where } h_s(F(\delta)) = X_{\tau'}.$$

Note: similar to the property of substitution, Lemma 2,

$$\mathcal{H}_a[\![(h_a, \phi)]\!](ss, \Delta, F) = \mathcal{H}_a[\![(h_a, \phi)]\!](ss, \Delta, F\mid_\Delta),$$

for any F. The $\mathcal{H}_a$ function either returns the empty set which means contradicting F or a singleton.

- the $\mathcal{ST}_a$ function (we use binding $\sigma_a = (g, pc, l, \omega, h, \phi)$):

$$\mathcal{ST}_a[\![\sigma_a]\!](F) = \{(\textit{sub-fun}(g, F), pc, \textit{sub-fun}(l, F), \textit{sub-seq}(\omega, F), h', \phi') \mid (h', \phi')$$
$$\in \mathcal{H}_a[\![(h, \phi)]\!](\textit{symbols}(\sigma_a), \textit{collect-sym-locs}(\sigma_a), F)\},$$

where *collect-sym-locs* takes a state and returns the set of symbolic locations that appear in the state. Since the return of $\mathcal{H}_a$ function can only be $\emptyset$ or a singleton, $\mathcal{ST}_a$ function returns $\emptyset$ or a singleton too.

Finally, the definition of $\gamma_a : \Sigma_a \to \mathcal{P}(\Sigma_s)$ is

$$\gamma_a(\sigma_a) = \bigcup_{\forall F \in \Pi} \mathcal{ST}_a[\![\sigma_a]\!](F).$$

Properties of γ_a

Definition 3. *A location i is a* <u>*legal*</u> *value for δ regarding to a SELA state $\sigma_a = (g, pc, l, \omega, h, \phi)$ if and only if the following conditions hold:*

1. *$\delta \in$ collect-sym-locs(σ_a);*

2. *$i \notin \operatorname{dom} h$ or $h(i)(\textsc{conc}) \uparrow$;*

3. *$(h', \phi') =$ init-loc-heap$(h, symbols(\sigma_a), \delta, i)$ with ϕ' is satisfiable.*

Lemma 3. *Let $\sigma_a \in \Sigma_a$ and $F \in \Pi$. Suppose $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. For any $(\delta, i) \in F$, if $\sigma'_a \in$ init-sym-loc(σ_a, δ, i) and i is a legal location for δ regarding to σ_a, then $\sigma_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$.*

Proof. Suppose $\sigma_a = (g_a, pc, l_a, \omega_a, h_a, \phi)$ and $\sigma_s = (g_s, pc, l_s, \omega_s, h_s, \phi_s)$. By the definition of *init-sym-loc*, $\sigma'_a = (\textit{sub-fun}_1(g_a, \delta, i), pc, \textit{sub-fun}_1(l_a, \delta, i), \textit{sub-seq}_1(\omega_a, \delta, i), h'_a, \phi')$, where $(h'_a, \phi') =$ *init-loc-heap*$(h_a, \phi, symbols(\sigma_a), \delta)i$. Since $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$, we have $g_s = \textit{sub-fun}(\textit{sub-fun}_1(g_a, \delta, i), F)$, $l_s = \textit{sub-fun}(\textit{sub-fun}_1(l_a, \delta, i), F)$, and $\omega_s = \textit{sub-seq}(\textit{sub-seq}_1(\omega_a, \delta, i), F)$ by Lemma 1. It remains to show that

$$(h_s, \phi_s) \in \mathcal{H}_a[\![(h'_a, \phi')]\!](symbols(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F).$$

We know that $(h_s, \phi_s) \in \mathcal{H}_a[\![(h_a, \phi)]\!](symbols(\sigma_a), \textit{collect-sym-locs}(\sigma_a), F)$. We will proceed by the definition of $\mathcal{H}_a$. Since σ'_a has one fewer symbolic location (δ) than σ_a, the predicate *well-mapped*$(symbols(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F)$ holds. Also it is easy to see that both *heap*$(\textit{collect-sym-locs}(\sigma'_a), h'_a, h_s, F)$ and $pc(\phi_s, \phi', h_s, F, \textit{collect-sym-locs}(\sigma'_a))$ hold. We conclude that $\sigma_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$ holds. □

Lemma 4. *Let $\sigma_a \in \Sigma_a$ and $F \in \Pi$. For any $(\delta, i) \in F$ where i is a legal location for δ regarding to σ_a, if $\sigma'_a \in$ init-sym-loc(σ_a, δ, i) and $\sigma_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$, then $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$.*

Proof. Similar to Lemma 3, the difficult part is to show that

$$(h_s, \phi_s) \in \mathcal{H}_a[\![(h_a, \phi)]\!](symbols(\sigma_a), \textit{collect-sym-locs}(\sigma_a), F).$$

We know that $(h_s, \phi_s) \in \mathcal{H}_a[\![(h'_a, \phi')]\!](\mathit{symbols}(\sigma'_a), \mathit{collect\text{-}sym\text{-}locs}(\sigma'_a), F)$. We will proceed by the definition of $\mathcal{H}_a$. Since σ_a has one more symbolic location (δ) than σ'_a and by i is legal for δ regarding to σ_a, the predicate $\mathit{well\text{-}mapped}(\mathit{symbols}(\sigma_a), \mathit{collect\text{-}sym\text{-}locs}(\sigma_a), F)$ holds. Also it is easy to see that both $\mathit{heap}(\mathit{collect\text{-}sym\text{-}locs}(\sigma_a), h_a, h_s, F)$ and $\mathit{pc}(\phi_s, \phi, h_s, F, \mathit{collect\text{-}sym\text{-}locs}(\sigma_a))$ hold. We conclude that $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$ holds. □

Lazier Kripke Structure

For any given method m, we have a set of global variables G and local variables L (ordered from $0..n$). We use Kripke structure $\mathcal{A} = (\Sigma_{\mathcal{A}}, I_{\mathcal{A}}, \longrightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ to model the state-space from the lazier initialization symbolic executions. The components are defined as follows:

- states, $\Sigma_{\mathcal{A}} = \Sigma_a \cup (\textsc{Exception} \times \Sigma_a) \cup (\textsc{Error} \times \Sigma_a)$.
- initial states,

  $$I_{\mathcal{A}} = \{\, (g_a, pc_{init}, l_a, nil, h_a, \{\textsc{True}\}) \mid \mathrm{dom}(g_a) = G \wedge \mathrm{dom}(l_a) = L \,\},$$

  and each local and global is initialized as follows: if its type is primitive, a symbolic primitive symbolic is created; otherwise, it is nondeterministically initialized as a fresh symbolic location or NULL.
- transition relation, $a \longrightarrow_{\mathcal{A}} a' \iff a \Rightarrow_{\mathcal{A}} a_2, a_2 \Rightarrow_{\mathcal{A}} a_3, \ldots, a_n \Rightarrow_{\mathcal{A}} a'$ for some $n \in \mathbb{N}$ with program counters of $a, a_2, \ldots, a_n$ are the same and the program counter of a and a' are different and the path condition of a' is satisfiable.
- labels, we do not use this part and thus they are ignored.

Similar to γ_s, function γ_a is trivially extended to $\gamma^*_a : \Sigma_{\mathcal{A}} \to \mathcal{P}(\Sigma_{\mathcal{S}})$ as

$$\gamma^*_a(a) = \begin{cases} \gamma_a(\sigma_a), & \text{if } a = \sigma_a \text{ for some } \sigma_a \in \Sigma_a; \\ \{\, (\textsc{Exception}, \sigma_s) \mid \sigma_s \in \gamma_a(\sigma_a) \,\}, & \text{if } a = (\textsc{Exception}, \sigma_a) \text{ for some } \sigma_a \in \Sigma_a; \\ \{\, (\textsc{Exception}, \sigma_s) \mid \sigma_s \in \gamma_a(\sigma_a) \,\}, & \text{if } a = (\textsc{Error}, \sigma_a) \text{ for some } \sigma_a \in \Sigma_a. \end{cases}$$

Simulation Relations

We introduce a relation $\mathcal{R}'$ between lazier symbolic states $\Sigma_{\mathcal{A}}$ and $\Sigma_{\mathcal{S}}$ as follows:

Definition 4. $\sigma_s \mathcal{R}' \sigma_a \iff \sigma_s \in \gamma_a^*(\sigma_a)$.

Clearly, for all $s_0 \in I_{\mathcal{S}}$, there exists a $a_0 \in I_{\mathcal{A}}$ such that $s_0 \mathcal{R}' a_0$.

Proposition 5. $\mathcal{S} \lhd_{\mathcal{R}'} \mathcal{A}$.

Proof. It is sufficient to show that for all $\sigma_s, \sigma_s' \in \Sigma_{\mathcal{S}}, \sigma_a \in \Sigma_{\mathcal{A}}$ if $\sigma_s \longrightarrow_{\mathcal{S}} \sigma_s'$ and $\sigma_s \mathcal{R}' \sigma_a$ then there exists $\sigma_a' \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma_a'$ and $\sigma_s' \mathcal{R}' \sigma_a'$. We will proceed with the rule induction on $\longrightarrow_{\mathcal{S}}$.

- Rule IF_ACMPEQ1-S: Let $\sigma_s = (g_s, pc, l_s, i :: j :: \omega_s, h_s, \phi)$. Then $i \neq j$ and $\sigma_s' = (g_s, next(pc), l_s, \omega_s, h_s, \phi)$. Suppose $\sigma_s \mathcal{R}' \sigma_a$, we need to show there exists $\sigma_a' \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma_a'$ and $\sigma_s' \mathcal{R}' \sigma_a'$. Since $\sigma_s \mathcal{R}' \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that σ_a has the form of $(g_a, pc, l_a, \delta_\tau :: \delta'_{\tau'} :: \omega_a, h_a, \phi')$ for some F with $\mathcal{V}'[\![\delta]\!](F) = i$, $\mathcal{V}'[\![\delta']\!](F) = j$, and $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. After taking the IF_ACMPEQ3-A rule, we get an invisible state $t_1 = (g_a', pc, l_a', i :: \delta'_{\tau'} :: \omega_a', h_a', \phi'')$ with $t_1 \in \textit{init-sym-loc}(\sigma_a, \delta, i)$. By Lemma 3, we have $\sigma_s \in \mathcal{ST}_a[\![t_1]\!](F)$. After taking the IF_ACMPEQ2-A rule, we get another invisible state $t_2 = (g_a'', pc, l_a'', i :: j :: \omega_a'', h_a'', \phi''')$ with $t_2 \in \textit{init-sym-loc}(t_1, \delta', j)$. By Lemma 3, we have $\sigma_s \in \mathcal{ST}_a[\![t_2]\!](F)$. Finally, we take the IF_ACMPEQ1-S rule and get $\sigma_a' = (g_a'', next(pc), l_a'', \omega_a'', h_a'', \phi''')$. Now it is sufficient to show that $\sigma_s' \in \gamma_a(\sigma_a')$. Clearly $\textit{sub-fun}(g_a'', F) = \textit{sub-fun}(g_a, F) = g_s$, $\textit{sub-fun}(l_a'', F) = \textit{sub-fun}(l_a, F) = l_s$, and $\textit{sub-seq}(\omega_a'', F) = \textit{sub-seq}(\omega_a, F) = \omega_s$ by applying Lemma 1 twice. It remains to show that
$(h_s, \phi) \in \mathcal{H}_a[\![(h_a'', \phi''')]\!](\textit{symbols}(\sigma_a'), \textit{collect-sym-locs}(\sigma_a'), F)$. Since $\textit{symbols}(\sigma_a') = \textit{symbols}(t_2)$ and $\textit{collect-sym-locs}(\sigma_a') = \textit{collect-sym-locs}(t_2) = \textit{collect-sym-locs}(\sigma_a) \setminus \{\delta, \delta'\}$, we get $(h_s, \phi) \in \mathcal{H}_a[\![(h_a'', \phi''')]\!](\textit{symbols}(\sigma_a'), \textit{collect-sym-locs}(\sigma_a'), F)$. Therefore, $\sigma_s' \in \gamma_a(\sigma_a')$.

- Rule GETFIELD3-S: Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then $\sigma_s' = (g_s, next(pc), l_s, \textsc{null} :: \omega_s, h_s', \phi)$ where $h_s(i) = Y$ and $h_s' = h_s[i \mapsto Y[f_\tau \mapsto \textsc{null}]]$. Suppose $\sigma_s \mathcal{R}' \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s\ \mathcal{R}'\ \sigma'_a$. Since $\sigma_s\ \mathcal{R}'\ \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that σ_a has the form of $(g_a, pc, l_a, \delta_{\tau'} :: \omega_a, h_a, \phi')$ for some F with $\mathcal{V}'[\![\delta]\!](F) = i$ and $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ with $t \in \textit{init-sym-loc}(\sigma_a, \delta, i)$. By Lemma 3, we have $\sigma_s \in \mathcal{ST}_a[\![t]\!](F)$. Finally, we take the rule GETFIELD3-S and get $\sigma'_a = (g'_a, \textit{next}(pc), l'_a, \text{NULL} :: \omega'_a, h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \text{NULL}]], \phi'')$. We need to show $\sigma'_s \in \gamma_a(\sigma'_a)$. By Lemma 1, $\textit{sub-fun}(g'_a, F) = \textit{sub-fun}(g_a, F) = g_s, \textit{sub-fun}(l'_a, F) = \textit{sub-fun}(l_a, F) = l_s$, and $\textit{sub-seq}(\omega'_a, F) = \textit{sub-seq}(\omega_a, F) = \omega_s$ hold. It is sufficient to show that $(h_s[i \mapsto h_s(i)[f_\tau \mapsto \text{NULL}]], \phi) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \text{NULL}]], \phi')]\!](\textit{symbols}(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F)$. Since $\textit{symbols}(\sigma'_a) = \textit{symbols}(t)$ and $\textit{collect-sym-locs}(\sigma'_a) = \textit{collect-sym-locs}(t) = \textit{collect-sym-locs}(\sigma_a \setminus \{\delta\})$, and $h'_a(i)(f) = h_s(i)(f) = \text{NULL}$, $(h_s[i \mapsto h_s(i)[f_\tau \mapsto \text{NULL}]], \phi) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \text{NULL}]], \phi'')]\!](\textit{symbols}(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F)$ by the definition of $\mathcal{H}_a$.

- Rule GETFIELD6-S Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then $\sigma'_s = (g_s, \textit{next}(pc), l_s, j :: \omega_s, h'_s, \phi')$ where $h_s(i) = Y^{m,n}$ and $h'_s = h_s[i \mapsto Y^{m,n}[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], \phi' = \phi \cup \{\tau' <: \tau\}$ where $Z_{\tau'} = \textit{new-sym}(\textit{symbols}(\sigma_s), m-1, k)$ and $j \notin \text{dom}\, h_s$. Suppose $\sigma_s\ \mathcal{R}'\ \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s\ \mathcal{R}'\ \sigma'_a$. Since $\sigma_s\ \mathcal{R}'\ \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that σ_a has the form of $(g_a, pc, l_a, \delta_{\tau'} :: \omega_a, h_a, \phi')$ for some F with $\mathcal{V}'[\![\delta]\!](F) = i$ and $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ with $t \in \textit{init-sym-loc}(\sigma_a, \delta, i)$. By Lemma 3, we get $\sigma_s \in \mathcal{ST}_a[\![t]\!](F)$. Finally, We can take GETFIELD3-S transition rule and get $\sigma'_a = (g'_a, \textit{next}(pc), l'_a, \delta_\tau :: \omega'_a, h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta'']], \phi'')$ where $\delta'' \notin \textit{collect-sym-locs}(t)$. Let $F' = F[\delta'' \mapsto j]$. Since δ'' is fresh in t, $\textit{sub-fun}(g'_a, F') = \textit{sub-fun}(g'_a, F) = g_s, \textit{sub-fun}(l'_a, F') = \textit{sub-fun}(l'_a, F) = l_s$, and $\textit{sub-seq}(\omega'_a, F') = \textit{sub-seq}(\omega'_a, F) = \omega_s$. It remains to show $(h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\}) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta'']], \phi'')]\!](\textit{symbols}(\sigma'_a), \textit{collect-sym-locs}(s'_a)$ Since we already have $(h_s, \phi) \in \mathcal{H}_a[\![(h'_a, \phi'')]\!](\textit{symbols}(t), \textit{collect-sym-locs}(t), F)$, according to the definition of $\mathcal{H}_a$ function, we only need to consider the extra elements: δ'', j, and Z. Since j is not in the domain of h_s, j is not in the domain of h'_a. So j is not in the domain of $h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta'']]$. We get *well-mapped*(*collect-sym-locs*(t) $\cup \{\delta''\}, h'_a[i \mapsto h'_a(i)[f \mapsto \delta'']], F'$). Since Z = *new-sym*(*symbols*(σ_s), $m-1, k$) and $F'(\delta'') = j$, we have *heap*(*collect-sym-locs*(t) $\cup \{\delta''\}, h'_a[i \mapsto h'_a(i)[f \mapsto \delta]], h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}]$). Since F' introduce a new entry (δ'', j) then $pc(\phi'', \phi \cup \{\tau' <: \tau\}, h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], F'$, *collect-sym-locs*($t$) $\cup \{\delta''\}$) holds. Thus $(h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\}) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta]], \phi'')]\!]$(*symbols*($\sigma'_a$), *collect-sym-locs*($\sigma'_a$), F') holds.

□

Next we define a relation.

Definition 5. $\mathcal{R}'_\bullet \subseteq \Sigma_{\mathcal{A}} \times \mathcal{P}(\Sigma_{\mathcal{S}})$*, as follows:*

$$\sigma_a \ \mathcal{R}'_\bullet \ S_s \iff \gamma^*_a(\sigma_a) = S_s$$

Clearly, $\mathcal{R}'_\bullet$ is left total. Since $\mathcal{R}'$ is right total, then for all σ_a, if $\sigma_a \ \mathcal{R}'_\bullet \ S_s$, then $S_s \neq \emptyset$. Furthermore, for any $\sigma_a \in I_{\mathcal{A}}$ and $\sigma_a \ \mathcal{R}'_\bullet \ S_s$, it is clear that $S_s \subseteq I_{\mathcal{S}}$ by the definition of γ_a function.

Proposition 6. $\mathcal{A} \lhd_{\mathcal{R}'_\bullet} \mathcal{P}(\mathcal{S})$.

Proof. It is sufficient to show that for all $\sigma_a \in \Sigma_{\mathcal{A}}, S_s \in \mathcal{P}(\Sigma_{\mathcal{S}})$ if $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma_a \ \mathcal{R}'_\bullet \ S_s$ and $\sigma'_a \ \mathcal{R}'_\bullet \ S'_s$ then $S_s \overset{\bullet}{\longrightarrow}_{\mathcal{S}} S'_s$.

We will prove by rule induction on transitions, $\longrightarrow_{\mathcal{A}}$.

- Rule `if_acmpeq`: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_\tau :: \delta'_{\tau'} :: \omega_a, h_a, \phi')$. Then by the definition of $\longrightarrow_{\mathcal{A}}$, the rule consists of three lazier symbolic transitions rules: IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking IF_ACMPEQ3-A rule, we get an invisible state $t_1 = (g'_a, pc, l'_a, i :: \delta'_{\tau'} :: \omega'_a, h'_a, \phi'')$ for some $i \in$ **Locs** and $t_1 \in$ *init-sym-loc*(σ_a, δ, i). Then after taking IF_ACMPEQ2-A rule, we get another invisible state $t_2 = (g''_a, pc, l''_a, i :: j :: \omega''_a, h''_a, \phi''')$ for some $j \in$ **Locs** and $t_2 \in$ *init-sym-loc*(t_2, δ', j). WLOG, suppose $i \neq j$ (the $i = j$ case is symmetric). Finally, we take IF_ACMPEQ1-S rule and get $\sigma'_a = (g''_a, next(pc), l''_a, \omega''_a, h''_a, \phi''')$. Suppose $\sigma_a \ \mathcal{R}'_\bullet \ S_s$ and $\sigma'_a \ \mathcal{R}'_\bullet \ S'_s$. We need to show that

$S_s \overset{\bullet}{\longrightarrow}_S S'_s$, that is, for any $\sigma'_s \in S'_s$, there exists some $\sigma_s \in S_s$ such that $\sigma_s \longrightarrow_S \sigma'_s$. Suppose $\sigma'_s \in S'_s$, that is, $\sigma'_s \in \gamma_a(s'_a)$. Then σ'_s must be in the form of $(g'_s, \mathit{next}(pc), l'_s, \omega'_s, h'_s, \phi)$ for some F and $\sigma'_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$. Define $\sigma_s = (g'_s, \mathit{next}(pc), l'_s, i :: j :: \omega'_s, h'_s, \phi)$. It is clear that $\sigma_s \longrightarrow_S \sigma'_s$. We only need to show $\sigma_s \in S_s$, that is, $\sigma_s \in \gamma_a(\sigma_a)$. Define $F' = F[\delta \mapsto i][\delta' \mapsto j]$. We will show $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F')$. Since δ and δ' do not appear in σ'_a, thus t_2, we have $\sigma_s \in \mathcal{ST}_a[\![t_2]\!](F')$ by Lemma 2 and property of $\mathcal{H}_a$. By applying Lemma 4 twice, we get $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F')$.

- Rule `getfield` f_τ: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_{\tau'} :: \omega_a, h_a, \phi')$ and $\tau \in \mathbf{Types}_{record}$. By the definition of $\longrightarrow_{\mathcal{A}}$, the transition consists of two lazier rules: GETFIELD1-A and (GETFIELD2-A, GETFIELD3-A, or GETFIELD1-S). After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ for some $i \in \mathbf{Locs}$ and $t = \mathit{init\text{-}sym\text{-}loc}(\sigma_a, \delta, i)$. WLOG, assume that f field is undefined in $h'_a(i)$. We take the GETFIELD2-A rule and get $\sigma'_a = (g'_a, \mathit{next}(pc), l'_a, \delta'_\tau :: \omega'_a, h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']], \phi'')$, where δ' is fresh in t.

  Suppose $\sigma_a \mathcal{R}'_\bullet S_s$ and $\sigma'_a \mathcal{R}'_\bullet S'_s$. We need to show that $S_s \overset{\bullet}{\longrightarrow}_S S'_s$, that is, for any $\sigma'_s \in S'_s$, there exists some $\sigma_s \in S_s$ such that $\sigma_s \longrightarrow_S \sigma'_s$. Suppose $\sigma'_s \in S'_s$, that is, $\sigma'_s \in \gamma_a(\sigma'_a)$. Then σ'_s must be in the form of $(g'_s, \mathit{next}(pc), l'_s, j :: \omega'_s, h'_s, \phi)$ for some F such that $F(\delta') = j$ and $\sigma'_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$. Define h_s as h'_s after following two operations:

  1. remove $h'_s(i)(f)$. So the f field of $h_s(i)$ becomes undefined.
  2. if no symbol in h_s has a field points to $h'_s(i)(f)$, then the entry at location $h'_s(i)(f)$ is removed from h_s.

  Define ϕ_s as satisfying $pc(\phi_s, \phi'', h_s, F, \mathit{collect\text{-}sym\text{-}locs}(t))$, so $\phi_s \cup \{\tau'' <: \tau\} = \phi$ where $Z_{\tau''} = h'_s(F(\delta'))$. Define $\sigma_s = (g'_s, \mathit{next}(pc), l'_s, i :: \omega'_s, h_s, \phi_s)$. We will first show $\sigma_s \in S_s$ and then $\sigma_s \longrightarrow_S \sigma'_s$. To show $\sigma_s \in S_s$, it suffices to show $\sigma_s \in \mathcal{ST}_a[\![t]\!](F)$ (then we can apply Lemma 4 with $F[\delta \mapsto i]$). Now we use the definition of $\mathcal{H}_a$ to show $(h_s, \phi_s) \in \mathcal{H}_a[\![(h'_a, \phi'')]\!](\mathit{symbols}(t), \mathit{collect\text{-}sym\text{-}locs}(t), F)$. Since pc predicate obviously holds by construction of ϕ_s, it suffices to show the well-mapped and heap predicates. Since h'_a has one fewer symbolic location (δ') than $h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']]$, *well-mapped*(*collect-sym-locs*(t), h'_a, F) holds. We will prove the *heap* predicate by a case analysis according to the freshness of $F(\delta')$:

- $F(\delta') \notin F(\textit{collect-sym-locs}(t) \cup \operatorname{dom} h'_a$: then the entry $(F(\delta'), h'_s(F(\delta'))$ is removed from h_s. Since $\textit{heap}(\textit{collect-sym-locs}(\sigma'_a), h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']], h'_s, F)$ holds and $\textit{collect-sym-locs}(\sigma'_a) - \textit{collect-sym-locs}(t) = \{\delta'\}$, we have $\textit{heap}(\textit{collect-sym-locs}(t), h'_a, F)$ holds.
- otherwise: the entry $(F(\delta'), h'_s(F(\delta'))$ is not removed from h_s by the definition of h_s. We are done because $\textit{heap}(\textit{collect-sym-locs}(\sigma'_a), h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']], h'_s, F)$ holds.

So we have proved $(h_s, \phi_s) \in \mathcal{H}_a[\![(h_a, \phi')]\!](\textit{symbols}(t), \textit{collect-sym-locs}(t), F)$. Thus $\sigma_s \in \mathcal{ST}_a[\![t]\!](F)$ holds and further, $\sigma_s \in S_s$. It remains to show that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$. There are two cases:

- $h_s(F(\delta'))$ is not defined: Since σ'_a has only δ' that is not in σ_a, so $h'_s(F(\delta'))$ is a fresh symbol. We can take the GETFIELD6-S rule and get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$.
- $h_s(F(\delta'))$ is defined: By the well-mappedness of h_a, $h_s(F(\delta'))(\textsc{conc})$ is not defined. So we can take the GETFIELD4-S rule and get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$.

□

Soundness and Completeness

Proposition 7 (Soundness)**.** *Given any symbolic trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s_n$ with $s_1 \in I_{\mathcal{S}}$, there is a corresponding lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ with $a_1 \in I_{\mathcal{A}}$ such that $s_k \mathcal{R}' a_k$ for all $1 \leq k \leq n$.*

Proof. We proceed by mathematical induction on n using Proposition 9. □

Proposition 8 (Completeness)**.** *Given any lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ with $a_1 \in I_{\mathcal{A}}$, there is a corresponding symbolic trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s_n$ such that $s_k \mathcal{R}' a_k$ for all $1 \leq k \leq n$ and $s_1 \in I_{\mathcal{S}}$.*

Proof. It is easy to show that there exists a trace in $\mathcal{P}(\mathcal{S})$, $S_1 \overset{\bullet}{\longrightarrow}_{\mathcal{S}} S_2 \overset{\bullet}{\longrightarrow}_{\mathcal{S}} \cdots \overset{\bullet}{\longrightarrow}_{\mathcal{S}} S_n$ such that $a_k\ \mathcal{R}_\bullet\ S_k$ for all $1 \le k \le n$ by mathematical induction on n using Proposition 6. Since $S_n \ne \emptyset$, we can pick a $s_n \in S_n$ and use the definition of $\overset{\bullet}{\longrightarrow}_{\mathcal{S}}$, then get the corresponding symbolic trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s_n$. □

5.3.3 Relative Soundness and Completeness of Symbolic Execution with Lazier# Initialization

Following the outline of Section 5.3.2, we relate SELB in Section 5.2.3 and SELA in Section 5.2.2. First, we define a function γ_b which takes a SELB state and returns all the SELA states that have the same shape and only symbolic references are initialized to either NULL or symbolic locations. Then we introduce binary relations between SELA states (power) and SELB states. Finally, we will prove the relative soundness and completeness of SELB with regard to SELA intra-procedurely.

Definition of γ_b

Let us first introduce a definition: The set of all symbolic reference environments

$$\Xi = \{\, G \mid G : \textbf{SymRefs} \to (\textbf{SymLocs} \cup \{\text{NULL}\})\,\}\,. \tag{5.2}$$

Then we define a function: *legal-env* $: \Sigma_b \to \mathcal{P}(\Xi)$ as

$$\begin{aligned} \textit{legal-env}(\sigma_b) = \{G \in \Xi \mid\ & G(\textit{collect-sym-refs}(\sigma_b)) \cap \textit{collect-sym-locs}(\sigma_b) = \emptyset \wedge \\ & \forall \hat{\delta}_1 \ne \hat{\delta}_2 \in \textit{collect-sym-refs}(\sigma_b). G(\hat{\delta}_1) = G(\hat{\delta}_2) \implies G(\hat{\delta}_1) = \text{NULL}\}, \end{aligned}$$

where *collect-sym-refs* collects all the symbolic references in a state.
And $\mathcal{ST}_b : \Sigma_b \times \Xi \to \Sigma_a$ as

$$\mathcal{ST}_b[\![\sigma_b]\!](G) = (\textit{sub-fun}(g, G), pc, \textit{sub-fun}(l, G), \textit{sub-seq}(\omega, G), \textit{sub-fun2}(h, G), \phi),$$

with binding $\sigma_b = (g, pc, l, \omega, h, \phi)$.

The definition of $\gamma_b : \Sigma_b \to \mathcal{P}(\Sigma_a)$ is

$$\gamma_b(\sigma_b) = \bigcup_{\forall G \in \mathit{legal\text{-}env}(\sigma_b)} \mathcal{ST}_b[\![\sigma_b]\!](G).$$

Properties of γ_b

Lemma 5. *Let $\sigma_b \in \Sigma_b$ and $G \in \mathit{legal\text{-}env}(\sigma_b)$. Suppose $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G)$ and $\sigma_a = (g_a, pc, l_a, \omega_a, h_a, \phi_a)$. For any $(\hat{\delta}, v) \in G$, if $\sigma'_b = \mathit{init\text{-}sym\text{-}ref}(\sigma_b, \hat{\delta}, v)$, then $(g_a, pc', l_a, \omega_a, h_a, \phi_a) = \mathcal{ST}_b[\![\sigma'_b]\!](G)$.*

Proof. Suppose $\sigma_b = (g_b, pc, l_b, \omega_b, h_b, \phi)$. By the definition of *init-sym-ref*, $\sigma'_b = (\mathit{sub\text{-}fun}_1(g_b, \hat{\delta}, v), pc', \mathit{sub\text{-}fun}_1(l_b, \hat{\delta}, v), \mathit{sub\text{-}seq}_1(\omega_b, \hat{\delta}, v), \mathit{sub\text{-}fun2}_1(h_b, \hat{\delta}, v), \phi)$. Since $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G)$, we have $g_a = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}_1(g_b, \hat{\delta}, v), G)$, $l_a = \mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}_1(l_b, \hat{\delta}, v), G)$, $\omega_a = \mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}_1(\omega_b, \hat{\delta}, v), G)$, and $h_a = \mathit{sub\text{-}fun2}(\mathit{sub\text{-}fun2}_1(h_b, \hat{\delta}, v)), G)$, by Lemma 1. We conclude that $(g_a, pc', l_a, \omega_a, h_a, \phi_a) \in \mathcal{ST}_b[\![\sigma'_b]\!](G)$ holds. □

Lemma 6. *Let $\sigma_b = (g_b, pc, l_b, \omega_b, \phi) \in \Sigma_b$ and $G \in \mathit{legal\text{-}env}(\sigma_b)$. For any $(\hat{\delta}, v) \in G$, if $\sigma'_b = \mathit{init\text{-}sym\text{-}ref}(\sigma_b, \hat{\delta}, v)$ and $(g_a, pc', l_a, \omega_a, h_a, \phi) = \mathcal{ST}_b[\![\sigma'_b]\!](G)$, then $(g_a, pc, l_a, \omega_a, h_a, \phi) = \mathcal{ST}_b[\![\sigma_b]\!](G)$.*

Proof. Proof is similar to Lemma 5. □

Lazier# Kripke Structure

For any given method m, we have a set of global variables *Globals* and local variables *Locals* (ordered from $0..n$). We use Kripke structure $\mathcal{B} = (\Sigma_\mathcal{B}, I_\mathcal{B}, \longrightarrow_\mathcal{B}, L_\mathcal{B})$ to model the state-space from the lazier# initialization symbolic executions. The components are defined as follows:

- states, $\Sigma_\mathcal{B} = \Sigma_b \cup (\textsc{Exception} \times \Sigma_b) \cup (\textsc{Error} \times \Sigma_b)$.
- initial states,

$$I_\mathcal{B} = \{\, (g_b, pc_{init}, l_b, nil, h_b, \{\textsc{True}\}) \mid \mathrm{dom}(g_b) = \mathit{Globals} \wedge \mathrm{dom}(l_b) = \mathit{Locals} \,\},$$

and each local and global is initialized as follows: if its type is primitive, a primitive symbol is created; otherwise, it is initialized as a fresh symbolic reference. Furthermore, h_b is the empty heap.

- transition relation, $b \longrightarrow_{\mathcal{B}} b' \iff b \Rightarrow_{\mathcal{B}} b_2, b_2 \Rightarrow_{\mathcal{B}} b_3, \ldots, b_n \Rightarrow_{\mathcal{B}} b'$ for some $n \in \mathbb{N}$ with program counters of $b, b_2, \ldots, b_n$ are the same and the program counters of b and b' are different and the path condition of b' is satisfiable.
- labels, we do not use this part and thus it is ignored.

Similar to γ_a, function γ_b is trivially extended to $\gamma_b^* : \Sigma_{\mathcal{B}} \to \mathcal{P}(\Sigma_{\mathcal{A}})$ as

$$\gamma_b^*(b) = \begin{cases} \gamma_b(\sigma_b), & \text{if } b = \sigma_b \text{ for some } \sigma_b \in \Sigma_b; \\ \{\, (\textsc{Exception}, \sigma_a) \mid \sigma_a \in \gamma_b(\sigma_b) \,\}, & \text{if } b = (\textsc{Exception}, \sigma_b) \text{ for some } \sigma_b \in \Sigma_b; \\ \{\, (\textsc{Exception}, \sigma_a) \mid \sigma_a \in \gamma_b(\sigma_b) \,\}, & \text{if } b = (\textsc{Error}, \sigma_b) \text{ for some } \sigma_b \in \Sigma_b. \end{cases}$$

Simulation Relations

We introduce a relation $\mathcal{R}''$ between lazier# symbolic states $\Sigma_{\mathcal{B}}$ and $\Sigma_{\mathcal{A}}$ as follows:

Definition 6. $\sigma_a \; \mathcal{R}'' \; \sigma_b \iff \sigma_a \in \gamma_b^*(\sigma_b)$.

Clearly, for all $a_0 \in I_{\mathcal{A}}$, there exists a $b_0 \in I_{\mathcal{B}}$ such that $a_0 \; \mathcal{R}'' \; b_0$.

Proposition 9. $\mathcal{A} \lhd_{\mathcal{R}''} \mathcal{B}$.

Proof. It is sufficient to show that for all $\sigma_a, \sigma_a' \in \Sigma_{\mathcal{A}}, \sigma_b \in \Sigma_{\mathcal{B}}$ if $\sigma_a \longrightarrow_{\mathcal{A}} \sigma_a'$ and $\sigma_a \; \mathcal{R}'' \; \sigma_b$ then there exists $\sigma_b' \in \Sigma_{\mathcal{B}}$ such that $\sigma_b \longrightarrow_{\mathcal{B}} \sigma_b'$ and $\sigma_a' \; \mathcal{R}'' \; \sigma_b'$. We will proceed with the rule induction on $\longrightarrow_{\mathcal{A}}$.

- Rule `if_acmpeq`: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_\tau :: \delta'_{\tau'} :: \omega_a, h_a, \phi')$. Then by the definition of $\longrightarrow_{\mathcal{A}}$, the rule consists of three lazier symbolic transitions rules: IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking IF_ACMPEQ3-A rule, we get an invisible state $t_1 = (g_a', pc, l_a', i :: \delta'_{\tau'} :: \omega_a', h_a', \phi'')$ for some $i \in$ **Locs** and $t_1 \in$ *init-sym-loc*(σ_a, δ, i). Then after taking IF_ACMPEQ2-A rule, we get another invisible state

$t_2 = (g''_a, pc, l''_a, i :: j :: \omega''_a, h''_a, \phi''')$ for some $j \in \mathbf{Locs}$ and $t_2 \in \mathit{init\text{-}sym\text{-}loc}(t_2, \delta', j)$. WLOG, suppose $i \neq j$ (the $i = j$ case is symmetric). Finally, we take IF_ACMPEQ1-S rule and get $\sigma'_a = (g''_a, \mathit{next}(pc), l''_a, \omega''_a, h''_a, \phi''')$. Suppose $\sigma_a \mathcal{R}'' \sigma_b$. We need to show that there exists any $\sigma'_b \in \Sigma_{\mathcal{B}}$ such that $\sigma_b \longrightarrow_{\mathcal{B}} \sigma'_b$. WLOG, suppose that $\sigma_b = (g_b, pc, l_b, \hat{\delta} :: \delta' :: \omega_b, h_b, \phi)$. Since $\sigma_a \mathcal{R}'' \sigma_b$, there exists $G \in \mathit{legal\text{-}env}(\sigma_b)$ such that $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G)$. Clearly $G(\hat{\delta}) = \delta$. We take rule IF_ACMEQ3-B and get a state $t'_0 = \mathit{init\text{-}sym\text{-}ref}(\sigma_b, \hat{\delta}, \delta)$ with stack $\delta :: \delta' :: \mathit{sub\text{-}seq}(\omega_b, \hat{\delta}, \delta)$. By Lemma 5, we get $\sigma_a \mathcal{R}'' t'_0$. Then we take IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S. We get $t'_1 = \mathit{init\text{-}sym\text{-}loc}(t'_0, \delta, i)$ after rule IF_ACMPEQ3-A, $t'_2 = \mathit{init\text{-}sym\text{-}loc}(t'_1, \delta', j)$ after rule IF_ACMPEQ2-A, and σ'_b after IF_ACMPEQ1-S. Since all the rules do not involve any symbolic references, it is clear that $t_1 \mathcal{R}'' t'_1$, and $t_2 \mathcal{R}'' t'_2$, and finally $\sigma'_a \mathcal{R}'' \sigma'_b$.

- Rule `getfield` f_τ: Suppose, WLOG, $\tau \in \mathbf{Types}_{non-prim}$ and $\sigma_a = (g_a, pc, l_a, i :: \omega_a, h_a, \phi)$ and $Y^{m,n} = h_a(i)$ and $Y(f) \uparrow$. Assume that rule GETFIELD2-A is taken. We get $\sigma'_a = (g_a, \mathit{next}(pc), l_a, \delta_\tau^{m-1,k} :: \omega_a, h_a[i \mapsto Y^{m,n}[f_\tau \mapsto \delta_\tau^{m-1,k}]], \phi)$ where δ is fresh. Suppose $\sigma_a \mathcal{R}'' \sigma_b$ and $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G)$ for some $G \in \mathit{legal\text{-}env}(\sigma_b)$. WLOG, assume $\sigma_b = (g_b, pc, l_b, i :: \omega_b, h_b, \phi)$. Clearly we have $X^{m,n} = h_b(i)$ for some X and $X(f) \uparrow$. After rule GETFIELD2-B, we get $\sigma'_b = (g_b, \mathit{next}(pc), l_b, \hat{\delta}_\tau^{m-1,k} :: \omega', h_b[i \mapsto Y^{m,n}[f_\tau \mapsto \hat{\delta}_\tau^{m-1,k}]], \phi)$ and $\hat{\delta}$ is fresh in σ_b. It is easy to see that $G[\hat{\delta} \mapsto \delta] \in \mathit{legal\text{-}env}(\sigma'_b)$. Thus we have $\sigma'_a = \mathcal{ST}_b[\![\sigma'_a]\!](G[\hat{\delta} \mapsto \delta])$, that is, $\sigma'_a \mathcal{R}'' \sigma'_b$.

□

Next we define a relation.

Definition 7. $\mathcal{R}''_\bullet \subseteq \Sigma_{\mathcal{B}} \times \mathcal{P}(\Sigma_{\mathcal{A}})$*, as follows:*

$$\sigma_b \mathcal{R}''_\bullet S_a \iff \gamma_b^*(\sigma_b) = S_a$$

Clearly, $\mathcal{R}''_\bullet$ is left total. Since $\mathcal{R}''$ is right total, then for all σ_b, if $\sigma_b \mathcal{R}''_\bullet S_a$, then $S_a \neq \emptyset$. Furthermore, for any $\sigma_b \in I_{\mathcal{B}}$ and $\sigma_b \mathcal{R}''_\bullet S_a$, it is clear that $S_a \subseteq I_{\mathcal{A}}$ by the definition of γ_b function.

Proposition 10. $\mathcal{B} \lhd_{\mathcal{R}''_\bullet} \mathcal{P}(\mathcal{A})$.

Proof. It is sufficient to show that for all $\sigma_b \in \Sigma_{\mathcal{B}}, S_a \in \mathcal{P}(\Sigma_{\mathcal{A}})$ if $\sigma_b \longrightarrow_{\mathcal{B}} \sigma'_b$ and $\sigma_b\ \mathcal{R}''_\bullet\ S_a$ and $\sigma'_b\ \mathcal{R}'_\bullet\ S'_a$ then $S_a \overset{\bullet}{\longrightarrow}_{\mathcal{A}} S'_a$.

We will prove by rule induction on transitions, $\longrightarrow_{\mathcal{B}}$.

- Rule `if_acmpeq`: Suppose, WLOG, $\sigma_b = (g_b, pc, l_b, \hat{\delta}_1 :: \hat{\delta}_2 :: \omega_b, h_b, \phi)$. Then by the definition of $\longrightarrow_{\mathcal{B}}$, the rule consists of five transitions rules: IF_ACMPEQ3-B, IF_ACMEQ2-B, IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking the IF_ACMPEQ3-B rule, we get an invisible state $t_1 = \textit{init-sym-ref}(\sigma_b, \hat{\delta}_1, \delta_1)$ and then IF_ACMPEQ2-B rule, we get $t_2 = \textit{init-sym-ref}(t_1, \hat{\delta}_2, \delta_2)$. Then by IF_ACMPEQ3-A rule, we get t_3 and by IF_ACMEQ2-A rule, we arrive at t_4 where δ_1 and δ_2 are fresh. WLOG, suppose we take IF_ACMPEQ1-S rule and get $\sigma'_b = (\textit{sub-fun}_1(\textit{sub-fun}_1(g_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2), \textit{next}(pc),$ $\textit{sub-fun}_1(\textit{sub-fun}_1(l_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2)$, $\textit{sub-seq}_1(\textit{sub-seq}_1(\omega_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2),$ $\textit{sub-fun2}_1(\textit{sub-fun2}_1(h_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2), \phi')$. Suppose $\sigma_b\ \mathcal{R}'_\bullet\ S_a$ and $\sigma'_b\ \mathcal{R}'_\bullet\ S'_a$. We need to show that $S_a \overset{\bullet}{\longrightarrow}_{\mathcal{S}} S'_a$, that is, for any $\sigma'_a \in S'_a$, there exists some $\sigma_a \in S_a$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$. Suppose $\sigma'_a \in S'_a$, that is, $\sigma'_a \in \gamma_b(\sigma'_b)$. Then σ'_a must be in the form of $(g'_a, \textit{next}(pc), l'_a, \omega'_a, h'_a, \phi)$ for some G and $\sigma'_a \in \mathcal{ST}_b[\![\sigma'_b]\!](G)$. Define $G' = G[\hat{\delta}_1 \mapsto \delta_1][\hat{\delta}_2 \mapsto \delta_2]$. Clearly $G' \in \textit{legal-env}(\sigma_b)$. Define $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G')$. We need to show that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$. After applying Lemma 5 twice, we get $\sigma_a = \mathcal{ST}_b[\![t_2]\!](G')$. Since σ_a only differs from t_2 by some symbolic references which are not operands of the instruction, σ_a can takes exactly the same rules and get σ'_a. We conclude that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$.

- Rule `getfield` f_τ: Suppose, WLOG, $\sigma_b = (g_b, pc, l_b, \hat{\delta}_{\tau'} :: \omega_b, h_b, \phi')$ and $\tau \in \textbf{Types}_{record}$. By the definition of $\longrightarrow_{\mathcal{B}}$, the transition multiple lazier# rules. The first one is GETFIELD1-B. WLOG, assume that the invisible state after GETFIELD1-B is $t_1 = (\textit{sub-fun}_1(g_b, \hat{\delta}, \delta), pc,$ $\textit{sub-fun}_1(l_b, \hat{\delta}, \delta), \textit{sub-seq}_1(\omega_b, \hat{\delta}, \delta), \textit{sub-fun2}_1(h_b, \hat{\delta}, \delta), \phi)$ for some fresh δ. Then rule GETFIELD1-A is taken and get an invisible state $t_2 = (g_2, pc, l_2, \omega_2, h_2, \phi_2) = \textit{init-sym-loc}(t, \delta, i)$ for some $i \in \textbf{Locs}$. WLOG, assume that f field is undefined in $h_2(i)$. We take the GETFIELD2-B rule and get $\sigma'_b = (g_2, \textit{next}(pc), l_2, \hat{\delta}'_\tau :: \omega_2, h_2[i \mapsto h_2(i)[f_\tau \mapsto \hat{\delta}']], \phi_2)$, where $\hat{\delta}'$ is fresh in t_2.

Suppose $\sigma_b\ \mathcal{R}''_\bullet\ S_a$ and $\sigma'_b\ \mathcal{R}''_\bullet\ S'_a$. We need to show that $S_a \overset{\bullet}{\longrightarrow}_{\mathcal{A}} S'_a$, that is, for any $\sigma'_a \in S'_a$, there exists some $\sigma_a \in S_a$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$. Suppose $\sigma'_a \in S'_a$, that is, $\sigma'_a \in \gamma_b(\sigma'_b)$. Then σ'_a must be in the form of $(g'_a, next(pc), l'_a, \delta' :: \omega'_a, h'_a, \phi)$ for some G such that $G(\hat{\delta}') = \delta'$ and $\sigma'_a \in \mathcal{ST}_b[\![\sigma'_b]\!](G)$. Define $G' = G[\hat{\delta} \mapsto \delta]$. Clearly $G' \in \textit{legal-env}(\sigma_b)$. Let $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G')$. Using Lemma 5, we get $\sigma_a = \mathcal{ST}_b[\![t_1]\!](G')$. Since t_1 only has more symbolic references than σ_a, rule GETFIELD1-A is applicable and get s'_2. Since $\hat{\delta}'$ is fresh in t_2 and $G(\hat{\delta}') = \delta'$, δ' is fresh in s'_2. Therefore, we can apply GETFIELD2-A and get σ'_a. We conclude that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$.

□

Soundness and Completeness

Proposition 11 (Soundness)**.** *Given any lazier symbolic trace* $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ *with* $a_1 \in I_{\mathcal{A}}$*, there is a corresponding lazier# symbolic trace* $b_1 \longrightarrow_{\mathcal{B}} b_2 \longrightarrow_{\mathcal{B}} \cdots \longrightarrow_{\mathcal{B}} b_n$ *with* $b_1 \in I_{\mathcal{B}}$ *such that* $a_k\ \mathcal{R}''\ b_k$ *for all* $1 \le k \le n$*.*

Proof. We proceed by mathematical induction on n using Proposition 9. □

Proposition 12 (Completeness)**.** *Given any lazier# symbolic trace* $b_1 \longrightarrow_{\mathcal{B}} b_2 \longrightarrow_{\mathcal{B}} \cdots \longrightarrow_{\mathcal{B}} b_n$ *with* $b_1 \in I_{\mathcal{B}}$*, there is a corresponding symbolic trace* $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ *such that* $a_k\ \mathcal{R}''\ b_k$ *for all* $1 \le k \le n$ *and* $a_1 \in I_{\mathcal{A}}$*.*

Proof. It is easy to show that there exists a trace in $\mathcal{P}(\mathcal{A})$, $S_1 \overset{\bullet}{\longrightarrow}_{\mathcal{A}} S_2 \overset{\bullet}{\longrightarrow}_{\mathcal{A}} \cdots \overset{\bullet}{\longrightarrow}_{\mathcal{A}} S_n$ such that $b_k\ \mathcal{R}''_\bullet\ S_k$ for all $1 \le k \le n$ by mathematical induction on n using Proposition 10. Since $S_n \neq \emptyset$, we can pick an $a_n \in S_n$ and use the definition of $\overset{\bullet}{\longrightarrow}_{\mathcal{A}}$, then get the corresponding lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$. □

Chapter 6

KUnit

In this chapter, we describe an analysis feedback extension, KUnit, for Kiasan. Section 6.1 presents the basics of generating and concretizing effective input states. Section 6.2 presents the JUnit test case generation, object graph visualization, and mock object generation for open systems.

Acknowledgments

This chapter is based on a paper titled " Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems" by Xianghua Deng, Robby, and John Hatcliff to appear in the Proceeding of Testing: Academic & Industrial Conference Practice And Research Techniques (TAIC PART) 2007 [24].

6.1 Foundations for KUnit

First we discuss the mathematical aspect of path coverage. Given a program P, we define a relation $\sim$: $i_1 \sim i_2$ for inputs i_1 and i_2 if and only if P will execute the same path, which has the same sequence of program counters, with inputs i_1 and i_2. This is well-defined because we only deal with sequential programs here. Intuitively, if i_1 and i_2 can not be differentiated by P, then $i_1 \sim i_2$. Clearly, $\sim$ is an equivalence relation. This equivalence relation $\sim$ partitions the input space. Since there is an one-to-one mapping between inputs and concrete traces, we can reason

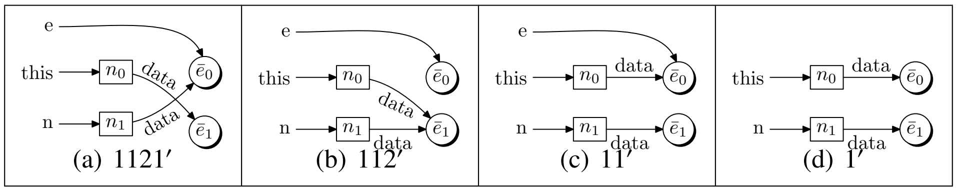


Figure 6.1: *Lazier# Backtrack States*

in term of traces instead of inputs. By the soundness of Kiasan, we know each concrete trace has a corresponding symbolic execution trace. Thus all the partitions are covered by the symbolic execution traces. Since we are only concerned with path coverage, it suffices to pick one trace from each partition of the input space by $\sim$. Thus for each symbolic trace, it suffices to generate a corresponding concrete trace.

As discussed above, to generate test cases, we need to concretize the input configuration for each symbolic path. This is not straightforward to do when using a lazy initialization algorithm because the initial symbolic state does not yet have sufficient heap information (recall that heap structure is discovered in an on-demand basis); one needs to reconstruct the heap structure at a method (unit) entry (pre-state) by using the information along the path to the method's exit points (post-states). This suggests that it might be possible to work backward from a post-state to the initial state. Fortunately, Kiasan is implemented on top of the extensible Bogor model checking framework [54] that provides a backtracking capability for "reverse" execution. However, the backtracking facility cannot be used as is; otherwise, one would get exactly the same initial state that the algorithm begins with.

We have formalized (and proved its *consistency* with Kiasan's algorithms) a modified reverse execution algorithm for KUnit that, given a path, produces a corresponding symbolic pre-state with sufficient heap information for concretization. In addition, we have formalized KUnit's concretization algorithm for producing a concrete state that *refines* a given symbolic pre-state. This section presents the intuition behind KUnit's algorithms. KUnit's formal semantics, consistency, and refinement proofs are presented in Chapter 7. In the next two sections, we describe the two main steps in our approach: (1) constructing effective input states that preserve materialized heap structures by using a modified backtracking algorithm, and (2) concretizing the effective input states into concrete input states.

6.1.1 Constructing the effective symbolic input state

Intuitively, for any symbolic execution trace $s_1 \Rightarrow \cdots \Rightarrow s_n$, we can give the path condition at the end state s_n to a constraint solver and get an instance of assignments of scalar values to primitive symbols. As mentioned above, we work backward from the end of each path and propagate the heap materialization to the initial state to obtain what we will call the *effective initial state*. Destructive updates to the heap complicate this further, however, these are naturally handled using the infrastructure of the backtracking facility employed in Bogor. To illustrate why a modified approach to backtracking is needed, consider that when using the normal backtracking approach, one would undo the materialization that resulted from lazy initialization and restore the current state to the pre-state of a transition. Since our goal is to *propagate* backward the materialization, we keep the lazily initialized heap objects intact in the undo step and do not "de-materialize" them. In addition, we keep the state's path condition intact. Although the path condition includes symbols (and their associated constraints) that are not mapped by variables at the input state, the symbols do not harm the process of concretizing values (one can remove the irrelevant symbols using transitive dependence information to determine if they are required for reasoning about variables at the input state). For example, consider Figure 6.1 that illustrates the construction of swap's expanded symbolic input state from State $1121'$ which is the same as 1121. From $1121'$, the algorithm backtracks to $112'$, then to $11'$, and finally to $1'$. Notice that when backtracking the first three statements of swap, we do not de-materialize lazily initialized objects. In addition, notice that State s' in Figure 6.1 refines State s in Figure 3.4 for $s \in \{1, 11, 112\}$.

6.1.2 Concretizing the effective symbolic input state

As mentioned above, when concretizing the effective symbolic input state, we use a constraint solver to get non-heap related value instantiations. For concretizing the remaining symbolic heap structures, there are four possible symbolic forms present in the state: (1) un-initialized fields, (2)

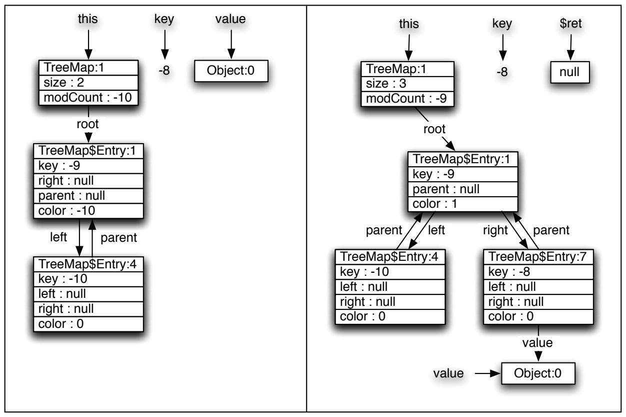


Figure 6.2: *Red-Black Tree* `put` *Pre-/Post-states*

symbolic locations, (3) symbolic references, and (4) symbolic objects. For uninitialized fields, we use default values according to the field's type. For each symbolic location/reference/object, we use a fresh object whose type satisfies the type constraints in the path condition of the state, and all of its fields are set to default values according to the fields' types. Note that this strategy works well with contract specifications because Kiasan embeds executable and effective (e.g., invariants transformed as parts of pre-/post-conditions) contracts along with the code [23]. That is, if a field is unconstrained even under the specified contracts, then using any value is fine. For the swap example, the symbolic objects n_0 and n_1 are concretized as Node objects. The uninitialized fields n_0.next and n_1.next are concretized with the NULL value, and the symbolic values $\bar{e}_0$ and $\bar{e}_1$ become fresh java.lang.Objects.

6.2 Test Cases and Object Graphs

6.2.1 Generating JUnit Test Cases

KUnit generates JUnit test cases using the concretization algorithm described in the previous section. Following the Design-by-Contract (DBC) methodology [46], each test case has the following structure:

assume effective pre–condition

invoke the method being tested

assert effective post–condition.

APIs. The *assume* uses post-order traversal of the heap to set up the input state. For each edge from o_1 to o_2 with label f (field), we use $o_1.f = o_2$. Since we use post-order, o_2 will be visited before o_1. The *assert* uses pre-order traversal of the heap to assert the equivalence of the current state and the input state. For each edge from o_1 to o_2 with label f (field), if o_2 has not been seen before, we use $o_2 = o_1.f$; otherwise, we use $assertEquals(o1.f, o_2)$ where assertEquals is a built-in method for asserting two arguments are equal in JUnit. The input and output states are built using Java reflection. While using reflection produces hard-to-read code, one can address this by adopting the JUnit-Objects [1] approach that uses XML descriptors to describe objects to be created.

6.2.2 Visualizing Effects using Object Graphs

To accompany each generated JUnit test case, KUnit visualizes a method's behavior by producing object graphs of the method's pre-/post-states. This is useful to provide a quick view of each of the test cases. It is interesting to note that as a consequence of the lazier# algorithm , the object graphs are *focused* on the heap objects that are accessed by Kiasan (this is in contrast to [14] that generates heap structures whose parts may not be accessed). Figure 6.2 presents a pre-/post-state pair for one test case of a put operation in the java.util.TreeMap red-black tree implementation. As can be observed, the value fields in the two entry objects at the pre-state are missing. We choose not to show such fields that are not accessed by the method (the JUnit test case input state has those fields' values equal to NULL, or default values in general). We believe this visualization has wide applicability for code inspection and understanding. For example, one can use this feature to try to understand the behavior of a fragment of "legacy" code by having KUnit quickly generate a variety of input/output pairs (which can be viewed as use cases of the code fragment). This is also

```java
public class Node<E> {
  private Node<E> next;

  private @NonNull E data;

  public Node(@NonNull E data) {
     this.data = data;
  }

  /*@requires acyclicOnNext() && ...;
    @ensures sorted(c)
     && elements().equals(\old(elements())); @*/
  public void sort(@NonNull Comparator<E> c) {
     ... c.compare(...) ...
  }
}

public interface Comparator<E> {
  ...

  //@ensures \result>0 || result==0 || \result<0;
  @Pure int compare(E e1, E e2); // transitive
}
```

(a) List Sort Example

```java
public class SortTest extends TestCase {
  int index = 2;

  public void testSort() throws Exception {
    Comparator c = new Comparator() {
      public int compare(Object o1, Object o2) {
        index--; ...
        switch (index) {
          case 1: return c1(this, o1, o2);
          case 0: return c0(this, o1, o2);
          default: throw new Error();
        }
      } ...
    };
    // build input state using reflection
    // call sort with c on input state
    // check post-condition
    ...
  }
  ...
  static int c0(Comparator c, Object o1, Object o2)
  { return -10; }
}
```

(b) Generated Test

Figure 6.3: *A List Sort and Generated Test with Mock Object (excerpts)*

useful for understanding how a given contract constrains the input states of a method (e.g., helpful when drafting contracts).

6.2.3 Closing Open Systems using Mock Objects

To generate test cases for open systems, one also needs to generate method implementations required to close the unit. For example, we need to generate an implementation of the Comparator interface passed as an argument to the sort method in Figure 6.3(a). This essentially amounts to generating mock objects.

In our approach, we generate mock objects by following the DBC methodology. Thus, each mock method *m* has the following basic structure:

assert m's *effective pre–condition*

assume m's *effective post–condition*

That is, the *assume* part simulates the effect of the mock method. For example, considering `compare`'s contract in Figure 6.3(a), we can initially design a mock object by simply having a method body that non-deterministically returns a negative, zero, or a positive integer.

However, one complicating factor when generating mock objects is that mock methods may be called multiple times using different calling contexts (even within a single test case). Thus, one has to summarize the behavior of the mock method for all the contexts. To address this, we leverage information about the sequence of method invocations from the symbolic execution path to create mock methods. That is, we remember the order of when a mock method is invoked, and we use the ordering number to index a behavior that simulates the effect of that particular invocation. We store the behavior of each invocation in a separate helper method. The overall mock method behavior is then memoized [47] by indexing the invocation to the helper methods. For example, Figure 6.3 illustrates a test case generated for the `sort` method in Figure 3.1. In this case, there are two objects in the input state, thus, there are two `compare` invocations. One is inside `sort` and another one is in its postcondition (i.e., `sorted`). For each invocation, KUnit creates a helper method such as `c0` (which returns -10, illustrating that in the last invocation, `compare` determines that the first argument is less than the second). Thus, the behavior of the `Comparator c` is spread to the helper methods `c0` and `c1` (not shown). To keep track of the ordering, we use an indexing counter, which decides which helper method should be invoked as implemented in the `compare` method in Figure 6.3(b).

Note that since `compare` is a pure method, we do not need to simulate side-effects using reflection. In general, side-effects are simulated using the same strategy used to build the input state, however, we do not need to rebuild the entire state. It is enough to construct parts of the state that are affected according to the specified contracts. Without contracts, KUnit assumes that there is no side-effect (i.e., best case). However, fresh symbolic (scalar/reference) values are used for return values (i.e., worst case). In short, KUnit's soundness is relative to user-supplied contracts.

The indexing strategy for memoizing behavior works well for demonstrating the paths explored by symbolic execution. We believe this approach already provides significant automation for developers as well as providing incentive to adopt the DBC methodology. That is, by using contracts instead of writing numerous test cases manually (which is labor intensive and difficult to get good coverage with), one can benefit from Kiasan's strong static checking, and KUnit's effective visualization and automatic test case generation. Moreover, contracts can serve as a formal basis for documentation purposes instead of using often ambiguous natural language descriptions.

Ideally, however, one would like to generate mock method behavior based on the context. That is, instead of memoizing the result value of each invocation, one would prefer to decide the result value based on the context. For example, instead of returning `-10` in `c0`, one would prefer to decide the return value based on `o1` and `o2`. This would make the test case insensitive to method invocation orderings, thus, it opens the possibility of reusing test cases when the method being analyzed (e.g., `sort`) is modified (e.g., when the sequence of mock method invocations maybe altered).

In the context of open systems with incomplete computation structures and weak contracts such as the one for `compare` that does not specify the relationship between return values/side-effects and its contexts, this cannot easily be done. One can employ a heuristic strategy, for example, using `Integer` objects when creating the input state of `sort` and leveraging the natural orderings of `Integer`s when creating an implementation of `compare`. However, the problem in general is difficult (i.e., heuristics can only be applied to a known set of interfaces), and we believe it is still a challenging research issue in open object-oriented systems. Even in practice, it is still hard to determine which test cases become stale after code modification, let alone generating test cases that are impervious to code modification.

In short, we believe that our approach serves the purpose for generating test cases. If invocation orderings are changed, the test cases may fail, but we encourage users to leverage contracts to statically re-check the modified code. The generated test cases themselves are evidence and provide feedback that our static analysis performs as it is supposed to (even when it does not find errors). We believe that our approach can be improved by employing heuristics strategy for commonly used interfaces, while further investigations are needed to address the general problem.

Chapter 7

Formalization of KUnit

In this chapter, we formalize the test input generation algorithms of KUnit and prove the path coverage of KUnit algorithms. Similar to the soundness and completeness proofs of Kiasan, the proofs have three parts: the first part is for the symbolic execution with lazy initialization shown in Section 7.1; the second part is for the symbolic execution with lazier initialization presented in Section 7.2; the last part is for the symbolic execution with lazier# initialization discussed in Section 7.3. Each part takes a similar procedure: first we define inverse (backtracking) rules for symbolic operational semantics rules; then we define a default concretization function; finally the input generation algorithm and the path coverage proof are given.

7.1 Input Generation Formalization for Symbolic Execution with Lazy Initialization

7.1.1 Backtracking Rules for Symbolic Execution with Lazy Initialization

Backtracking rules are the inverses of transition rules $\Rightarrow_{\mathcal{S}}$ (suppose we take one path at a time). Then after any transition $\sigma_1 \Rightarrow_{\mathcal{S}} \sigma_2$, we can apply the corresponding backtracking rule which takes σ_2 and returns σ_1. The simple way to achieve the backtracking functionality is to save the old state σ_1. But this is inefficient in practice, since there is usually few changes from σ_1 to σ_2. Instead, only the changes (δ) are stored in practice. For simplicity, we just take the simple approach and define backtracking rules as $(\Sigma_s \times \Sigma_s) \Rightarrow_{\mathcal{S}}^{-1} \Sigma_s$. Specifically, assume $\sigma_1 \Rightarrow_{\mathcal{S}} \sigma_2$ by rule FOO-S, then its backtracking rule FOO-S-BACK is $\langle\sigma_1, \sigma_2\rangle \Rightarrow_{S}^{-1} \sigma_1$.

In order to generate concrete test inputs, we modify two kinds of backtracking rules:

1. rules that involve lazy initialization, such as `getfield` and `iaload`. The lazy initialized fields/indexes are kept in the return state.

2. rules that add new constraints into path condition. The added constraints are kept in the return state.

Formally, each backtracking rule is in the format of

$$\frac{premises}{\langle\sigma_1, \sigma_2\rangle \Rightarrow_{S}^{-1} \sigma_3},$$

where $\sigma_1, \sigma_2, \sigma_3 \in \Sigma_{S}$. We call σ_1 the first state; σ_2 the second state; σ_3 the return state.

Figure 7.1 presents the backtracking rules:

- rule IADD-S-BACK is the corresponding backtracking rule for IADD-S. The IADD-S-BACK replaces the top element of the stack of the second state with the two top elements of the stack of the first state; and returns the changed second state. It can be seen as popping the result of addition and pushing the two operands to the stack.

- rules IF_ICMPLT-S-T-BACK and IF_ICMPLT-S-F-BACK are the corresponding backtracking rules for IF_ICMLT-S. There are two backtracking rules because rule IF_ICMLT-S has two possible end states depending on whether the true branch or false branch is taken. So when the top element is less than the the element below it in the stack of the first state (condition is true), IF_ICMPLT-S-T-BACK applies and returns the second state (except the programming counter using the first state) with two operands on the top of the first state being pushed onto the stack of the second state. Otherwise, rule IF_ICMPLT-S-F-BACK applies.

- rule NEW-S-BACK is the corresponding backtracking rule for NEW-S. The backtracking rule removes the newly created heap entry indexed by the top of the stack of the second state.

We use the bindings, $\sigma = (g, pc, l, \omega, h, \phi)$ and $\sigma' = (g', pc', l', \omega', h', \phi')$, for all the backtracking rules.

$$\text{IADD-S-BACK}\ \frac{code(pc) = \texttt{iadd} \qquad \omega = v_1 :: v_2 :: \omega_1 \qquad \omega' = v' :: \omega_2 \qquad pc' = next(pc)}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', v_1 :: v_2 :: \omega_2, h', \phi')}$$

$$\text{IF\_ICMPLT-S-T-BACK}\ \frac{\begin{array}{c} code(pc) = \texttt{if\_icmplt}\ pc_1 \\ \omega = v_1 :: v_2 :: \omega_1 \qquad v_2 < v_1 \in \phi' \qquad pc' = pc_1 \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', v_1 :: v_2 :: \omega', h', \phi')}$$

$$\text{IF\_ICMPLT-S-F-BACK}\ \frac{\begin{array}{c} code(pc) = \texttt{if\_icmplt}\ pc_1 \\ \omega = v_1 :: v_2 :: \omega_1 \qquad v_2 \nless v_1 \in \phi' \qquad pc' = next(pc) \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', v_1 :: v_2 :: \omega', h', \phi')}$$

$$\text{NEW-S-BACK}\ \frac{code(pc) = \texttt{new}\ \tau \qquad \omega' = i' :: \omega_2 \qquad pc' = next(pc)}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l, \omega_2, h' \setminus \{(i', h'(i'))\}, \phi')}$$

$$\text{GETFIELD-S-BACK}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \\ \omega = i :: \omega_1 \qquad pc' = next(pc) \qquad \omega' = v' :: \omega_2 \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', i :: \omega_2, h', \phi')}$$

$$\text{GETFIELD-S-Ex-BACK}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = \text{NULL} :: \omega_1 \\ \sigma' = \text{NullPointerException}\ , (g', pc', l', \omega', h', \phi') \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', \text{NULL} :: \omega', h', \phi')}$$

$$\text{PUTFIELD-S-BACK1}\ \frac{\begin{array}{c} code(pc) = \texttt{putfield}\ f \\ \omega = v :: i :: \omega_1 \qquad pc' = next(pc) \qquad h'(i)(f) = v \qquad h(i)(f) \downarrow \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', v :: i :: \omega', h'[i \mapsto h'(i)[f \mapsto h(i)(f)]], \phi')}$$

$$\text{PUTFIELD-S-BACK2}\ \frac{\begin{array}{c} code(pc) = \texttt{putfield}\ f \\ \omega = v :: i :: \omega_1 \qquad pc' = next(pc) \qquad h'(i)(f) = v \qquad h(i)(f) \uparrow \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', \omega', h'[i \mapsto h'(i) \setminus \{(f, h'(i)(f))\}], \phi')}$$

$$\text{PUTFIELD-S-Ex-BACK}\ \frac{\begin{array}{c} code(pc) = \texttt{putfield}\ f \qquad \omega = v :: \text{NULL} :: \omega_1 \\ \sigma' = \text{NullPointerException}\ , (g', pc', l', \omega', h', \phi') \end{array}}{\langle \sigma, \sigma' \rangle \Rightarrow_S^{-1} (g', pc, l', v :: \text{NULL} :: \omega', h', \phi')}$$

Figure 7.1: *Backtracking Rules for Symbolic Execution with Lazy Initialization*

- rule GETFIELD-S-BACK is the corresponding backtracking rule for rules GETFIELD[1..6]-S. This is because we preserve the results of lazy initialization.

- rule GETFILED-S-Ex-BACK is the corresponding backtracking rule for GETFIELD7-S which throws a NullPointerException.

- rules PUTFIELD-S-BACK1 and PUTFIELD-S-BACK2 are the corresponding backtracking rules for the rule PUTFIELD1-S. Both backtracking rules undo the writing to a field by rule PUTFIELD1-S. When the field is defined in the first state, PUTFIELD-S-BACK1 applies; otherwise, PUTFIELD-S-BACK2 applies.

- rule PUTFIELD-S-Ex-BACK is the corresponding backtracking rule for PUTFIELD2-S.

Lemma 7. *For any $\sigma_1 \longrightarrow_S \sigma_2$, there exists some state σ_3 such that $\langle \sigma_1, \sigma_2 \rangle \Rightarrow_S^{-1} \sigma_3$, and we have $\sigma_3 \longrightarrow_S \sigma_2$ and the transition rule is not one of the rules that involve lazy initialization (such as* GETFIELD3-S, GETFIELD4-S, GETFIELD5-S, GETFIELD6-S, *etc.).*

Proof. We will proceed with rule induction on operational semantic rules of SEL, $\longrightarrow_S$ (we only present representative rules):

- Rule IADD-S, assume that $\sigma_1 = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$. Then $\sigma_2 = (g_s, next(pc), l_s, Z :: \omega_s, h_s, \phi \cup \{Z = v_1 + v_2\})$ where Z is fresh. After applying rule IADD-S-BACK, we get $\sigma_3 = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi \cup \{Z = v_1 + v_2\})$. Clearly, $\sigma_3 \longrightarrow_S \sigma_2$ by rule IADD-S. For any E, T that satisfy $\phi \cup \{Z = v_1 + v_2\}$, we have $\mathcal{V}_s[\![v_1]\!](T, E) + \mathcal{V}_s[\![v_2]\!](T, E) = \mathcal{V}_s[\![Z]\!](T, E)$. The other components of σ_2 and σ_3 are the same.

- Rule IF_ICMLT-S, let $\sigma_1 = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$. WLOG, assume that the false branch is taken. Then $\sigma_2 = (g_s, next(pc), l_s, \omega_s, h_s, \phi \cup \{v_2 \geq v_1\})$. After applying the IF_ICMLT-F-S-BACK rule, we get $\sigma_3 = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi \cup \{v_2 \geq v_1\})$. Clearly, $\sigma_3 \longrightarrow_S \sigma_2$ by rule IF_ICMLT-S (the true branch is infeasible).

- Rule NEW-S, let $\sigma_1 = (g_s, pc, l_s, \omega_s, h_s, \phi)$. Then $\sigma_2 = (g_s, next(pc), l_s, i :: \omega_s, h_s[i \mapsto new\text{-}obj(symbols(\sigma_1), \tau)], \phi)$ where i is fresh in h_s. After applying the rule NEW-S-BACK, we get $\sigma_3 = (g_s, pc, l_s, \omega_s, h_s, \phi) = \sigma_1$. Clearly, $\sigma_3 \longrightarrow_S \sigma_2$.

- Rule GETFIELD3-S, let $\sigma_1 = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then f is not defined in $h_s(i)$ and $\sigma_2 = (g_s, next(pc), l_s, \text{NULL} :: \omega_s, h'_s, \phi)$ where $h'_s = h_s[i \mapsto h_s(i)[f_\tau \mapsto \text{NULL}]]$. After applying rule GETFIELD-S-BACK, we get $\sigma_3 = (g_s, pc, l_s, i :: \omega_s, h'_s, \phi)$. Clearly $\sigma_3 \longrightarrow_S \sigma_2$ by rule GETFIELD1-S.

- Rule GETFIELD6-S, $\sigma_1 = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then f is not defined in $h_s(i)$ and $\sigma_2 = (g_s, next(pc), l_s, j :: \omega_s, h'_s, \phi')$ where $h_s(i) = Y^{m,n}$, $h'_s = h_s[i \mapsto Y^{m,n}[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], \phi' = \phi \cup \{\tau' <: \tau\}$ for $Z_{\tau'} = new\text{-}sym(symbols(\sigma_s), m-1, k)$ and $j \notin \operatorname{dom} h_s$. After applying rule GETFIELD-S-BACK, we get $\sigma_3 = (g_s, pc, l_s, i :: \omega_s, h'_s, \phi')$. Clearly $\sigma_3 \longrightarrow_S \sigma_2$ by rule GETFIELD1-S.

□

We can have an even stronger property of the backtracking rules:

Lemma 8. *Given backtracking rule $\langle \sigma_1, \sigma_2 \rangle \Rightarrow_S^{-1} \sigma_3$ for some state $\sigma_1, \sigma_2, \sigma_3$. Then $\sigma_3 \longrightarrow_S \sigma_2$ and the transition rule is not one of the rules that involve lazy initialization (such as* GETFIELD3-S, GETFIELD4-S, GETFIELD5-S, GETFIELD6-S, *etc.).*

Proof. It can be shown by using rule induction of $\Rightarrow_S^{-1}$. □

7.1.2 Default Concretization Function for Symbolic Execution with Lazy Initialization

First, we define a default concretization function $default\text{-}concr : \Gamma \times Env \times \Sigma_S \to \Sigma_C$. The goal of *default-concr* is to generate a default concrete state for a SEL state given E and T satisfying the path condition of the SEL state. Intuitively, the concrete state is generated by substituting symbolic values with E and all the undefined fields are initialized with default values.

Then we introduce some semantic functions to facilitate the definition of *default-concr*. $\mathcal{O}_d$ maps a symbolic object/array into a concrete object or array. $\mathcal{H}_d$ maps a symbolic heap into a concrete heap by mapping each symbol in its range to a concrete object using $\mathcal{O}_d$.

$$\mathcal{O}_d : \mathbf{Symbols}_{non-prim} \to ((\Gamma \times Env) \to \mathbf{Symbols}_{non-prim});$$
$$\mathcal{H}_d : \mathbf{Heaps}_s \to ((\Gamma \times Env) \to \mathbf{Heaps}_c).$$

Here are the definitions (e is the identity permutation of locations):

- the $\mathcal{O}_d$ function:

$$\mathcal{O}_d[\![X_\tau]\!](T, E) = X'_{\tau'},$$

  where $\tau' = sub(\tau, T)$ and if $\tau' \in \mathbf{Types}_{record}$: for all $f_{\tau''} \in fields(\tau')$,

$$X'(f_{\tau''}) = \begin{cases} \mathcal{V}_s[\![X(f)]\!](E, e) & \text{if } X(f)\downarrow \\ default(\tau'') & \text{otherwise} \end{cases}$$

  if $\tau' \in \mathbf{Types}_{array}$: $X'(\textsc{len}) = \mathcal{V}_s[\![X(\textsc{len})]\!](E, e) \wedge \forall \iota \in acc\text{-}idx(X). X'(\mathcal{V}_s[\![\iota]\!](E, e)) = \mathcal{V}_s[\![X(\iota)]\!](E, e)$ and $\forall (0 \le m < X'(\textsc{len}) \wedge m \notin \{\, \mathcal{V}_s[\![\iota]\!](E, e) \mid \iota \in acc\text{-}idx(X)\,\})$.

$$X'(m) = \begin{cases} X(\textsc{def}) & \text{if } X(\textsc{def})\downarrow \\ default(\tau'') & \text{otherwise} \end{cases}$$

  where τ'' is the element type of τ'.

- the $\mathcal{H}_d$ function:

$$\mathcal{H}_d[\![h_s]\!](T, E) = h_c$$

  where

$$\operatorname{dom} h_s = \operatorname{dom} h_c \wedge \forall (i, X) \in h_s.(i, \mathcal{O}_d[\![X]\!](T, E)) \in h_c.$$

Finally the *default-concr* function:

$$\begin{aligned} dc(T, E, (g, pc, l, \omega, h, \phi)) = (&sub\text{-}fun(sub\text{-}fun(g, E), e), pc, \\ &sub\text{-}fun(sub\text{-}fun(l, E), e), sub\text{-}seq(sub\text{-}seq(\omega, E), e), \mathcal{H}_d[\![h]\!](T, E), \textsc{True}). \end{aligned}$$

$$\textit{default-concr}(T, E, s) = \begin{cases} dc(T, E, \sigma_s) & \text{if } s = \sigma_s; \\ (\textsc{Exception}, dc(T, E, \sigma_s)) & \text{if } s = (\textsc{Exception}, \sigma_s); \\ (\textsc{Error}, dc(T, E, \sigma_s)) & \text{if } s = (\textsc{Error}, \sigma_s). \end{cases}$$

It is easy to show that for all E, T that satisfy the path condition, ϕ of σ_s, $\textit{default-concr}(T, E, \sigma_s) \in \gamma_s(\sigma_s)$.

Lemma 9. *Suppose $\sigma_1 \longrightarrow_{\mathcal{S}} \sigma_2$ and the transition rule is not one of the lazy initialization rules. For any E, T satisfy ϕ_2 of σ_2, $\textit{default-concr}(T, E, \sigma_1) \longrightarrow_{C} \textit{default-concr}(T, E, \sigma_2)$*

Proof. Proceed by rule induction on $\longrightarrow_{\mathcal{S}}$. □

7.1.3 Input Generation Algorithm and Proof

Given any trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots s_n \longrightarrow_{\mathcal{S}} s_{n+1}$. Define a sequence of states s'_i for $1 \leq i \leq n + 1$ as

$$\begin{aligned} & s'_{n+1} = s_{n+1}, \\ & \langle s_n, s'_{n+1} \rangle \Rightarrow_{\mathcal{S}}^{-1} s'_n, \\ & \vdots \\ & \langle s_2, s'_3 \rangle \Rightarrow_{\mathcal{S}}^{-1} s'_2, \\ & \langle s_1, s'_2 \rangle \Rightarrow_{\mathcal{S}}^{-1} s'_1. \end{aligned}$$

The applicability of backtracking rules is shown by Lemma 12. Then we apply the default concretization function for s'_i, where $1 \leq i \leq n + 1$ for any $E, T \vDash \phi_{n+1}$ with ϕ_{n+1} is the path condition of s_{n+1}. And we get $c_1 = \textit{default-concr}(T, E, s'_1), \ldots, c_{n+1} = \textit{default-concr}(T, E, s'_{n+1})$.

Proposition 13. *$c_1 \longrightarrow_C c_2 \cdots \longrightarrow_C c_{n+1}$ and $c_i \in \gamma_s(s_i)$ for all $1 \leq i \leq n + 1$;*

To prove this main theorem, we need one additional definition and some lemmas.

Definition 8. *Relation $\prec_{\mathcal{S}}$: $\Sigma_{\mathcal{S}} \times \Sigma_{\mathcal{S}}$ as $s_1 \prec_{\mathcal{S}} s_2$ if and only if s_1 is similar to s_2 except that there may be some fields of symbolic objects/arrays in s_1 are defined but not in s_2.*

Precisely, $s_1 \prec_{\mathcal{S}} s_2$ for $s_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1)$ and $s_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2)$ if and only if

$$g_1 = g_2 \wedge pc_1 = pc_2 \wedge l_1 = l_2 \wedge \omega_1 = \omega_2 \wedge \mathit{refineheap}(pc_1, h_1, h_2) \wedge \phi_1 \supseteq \phi_2,$$

where $\mathit{refineheap}(pc_1, h_1, h_2)$ if and only if following conditions hold

1. $\operatorname{dom} h_2 \subseteq \operatorname{dom} h_1$;
2. for all $i \in \operatorname{dom} h_2.(\forall \iota. h_2(i)(\iota) \downarrow \implies (h_1(i)(\iota) \downarrow \wedge h_2(i)(\iota) = h_1(i)(\iota)) \wedge$
   $h_2(i)(\textsc{conc}) \downarrow \iff h_1(i)(\textsc{conc}) \downarrow$;
3. $\mathit{code}(pc_1) = \texttt{getfield} f \implies \omega_2 = i :: \omega_2' \wedge h_1(i)(f) \downarrow$ [1];
4. for all $i \in \operatorname{dom} h_1 \setminus \operatorname{dom} h_2$. $h_1(i)(\textsc{conc}) \uparrow \wedge \forall \iota \in \mathit{acc\text{-}idx}(h_1(i)).\mathit{non\text{-}concrete}(h_1, i, \iota)$;
5. for all $i \in \operatorname{dom} h_2$. for all $\iota.(h_1(i)(\iota) \downarrow) \wedge (h_2(i)(\iota) \uparrow) \implies \mathit{non\text{-}concrete}(h_1, i, \iota)$,

where

$$\mathit{non\text{-}concrete}(h_1, i, \iota) \equiv h_1(i)(\iota) = j \text{ for some } j \in \mathbf{Locs} \text{ implies } h_1(j)(\textsc{conc}) \uparrow .$$

Clearly, $\gamma_s(s_1) \subseteq \gamma_s(s_2)$.

Lemma 10. *Suppose $\sigma_1 \longrightarrow_{\mathcal{S}} \sigma_2$. Let σ_3 be the outcome of backtracking from σ_2, that is, $\langle \sigma_1, \sigma_2 \rangle \Rightarrow_{\mathcal{S}}^{-1} \sigma_3$. Then $\sigma_3 \prec_{\mathcal{S}} \sigma_1$.*

Proof. This can be shown easily by rule induction on $\longrightarrow_{\mathcal{S}}$. □

Lemma 11. *Suppose we have $\sigma_4 \prec_{\mathcal{S}} \sigma_2$ and $\sigma_1 \longrightarrow_{\mathcal{S}} \sigma_2$. Let σ_3 be the outcome of backtracking from σ_4, that is, $\langle \sigma_1, \sigma_4 \rangle \Rightarrow_{\mathcal{S}}^{-1} \sigma_3$. Then $\sigma_3 \prec_{\mathcal{S}} \sigma_1$.*

Proof. We will prove by rule induction of SEL semantics rules: $\longrightarrow_{\mathcal{S}}$. We will use the default bindings: $\sigma_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1), \sigma_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2), \sigma_3 = (g_3, pc_3, l_3, \omega_3, h_3, \phi_3)$, and $\sigma_4 = (g_4, pc_4, l_4, \omega_4, h_4, \phi_4)$.

[1] Similar properties should hold for `iaload` and `aaload`. For the purpose of simpler presentation, it is not listed.

- Rule IADD-S: Since $\sigma_1 \Rightarrow_{\mathcal{S}} \sigma_2$ by rule IADD-S, we know $\omega_1 = v_1 :: v_2 :: \omega'_1$, $code(pc_1) =$ `iadd`, $\omega_2 = v' :: \omega'_1$, $pc_2 = next(pc_1)$, $g_1 = g_2$, $l_1 = l_2$, $h_1 = h_2$, and $\phi_1 \subseteq \phi_2$. Since $\sigma_4 \prec_{\mathcal{S}} \sigma_2$, we have $g_4 = g_2, pc_2 = pc_4, l_4 = l_2, \omega_2 = \omega_4, \mathit{refineheap}(pc_4, h_4, h_2), \phi_2 \subseteq \phi_4$. By rule IADD-S-BACK, we get $\sigma_3 = (g_4, pc_1, l_4, v_1 :: v_2 :: \omega'_4, h_4, \phi_4)$, where $\omega_4 = v_4 :: \omega'_4$. Then we get $g_3 = g_4 = g_2 = g_1, pc_3 = pc_1, l_3 = l_4 = l_2 = l_1, \mathit{refineheap}(pc_3, h_3, h_1), \phi_1 \subseteq \phi_2 \subseteq \phi_4 = \phi_3$. Thus we conclude that $\sigma_3 \prec_{\mathcal{S}} \sigma_1$.

- Rule IF_ICMPLT-S: WLOG, assume that false branch is taken. Similar to the IADD-S-BACK case.

- Rule NEW-S: The proof of $g_3 = g_1, pc_3 = pc_1, \omega_3 = \omega_1$, and $\phi_1 \subseteq \phi_3$ is similar to the IADD-S case. The interesting part is to show that $\mathit{refineheap}(pc_3, h_3, h_1)$. We know $\mathit{refineheap}(pc_4, h_4, h_2)$, $h_3 = h_4 \setminus \{(i', h_4(i'))\}$ and $h_1 = h_2 \setminus \{(i', h_2(i'))\}$. First of all, we need to show that σ_3 is a well-defined SEL state, that is, i' is not referred in σ_3. Since $g_3 = g_4 = g_2 = g_1$ and i' is fresh in σ_1, g_3 does not refer to i'. Similarly, l_3 and ω_3 do not refer to i'. Since $\mathit{refineheap}(pc_4, h_4, h_2)$, $h_4 \setminus \{(i', h_4(i'))\}$ does not refer to i' by Properties 4 and 5 of *refineheap*. Thus σ_3 is well-defined SEL state. Then it is clear that $\mathit{refineheap}(pc_3, h_3, h_1)$. Thus we conclude $\sigma_3 \prec_{\mathcal{S}} \sigma_1$.

- Rule GETFIELD2-S: Similar to the Rule NEW-S case, the interesting part is to show that $\mathit{refineheap}(pc_3, h_3, h_1)$. Suppose $\omega_1 = i :: \omega'_1$ and $code(pc_1) =$ `getfield` f. Then $h_2(i)(f)\downarrow$. By $\sigma_4 \prec_{\mathcal{S}} \sigma_2$, $h_4(i)(f)\downarrow$. Using the GETFIELD-S-BACK rule, we get $h_4 = h_3$, in particular, $h_3(i)(f)\downarrow$. Thus we get $\mathit{refineheap}(pc_3, h_3, h_1)$. The other GETFIEDx-S rules are similar.

- Rule PUTFIELD1-S: Again, the interesting part is to show $\mathit{refineheap}(pc_3, h_3, h_1)$. Suppose $code(pc_1) =$ `putfield` f, $\omega_1 = v :: i :: \omega'_1$, and $h_1(f)\uparrow$. From PUTFIELD1-S rule, we know $h_2(i)(f) = v$. Since $\sigma_4 \prec_{\mathcal{S}} \sigma_2$, $h_4(i)(f) = v$ and v can not point to $\operatorname{dom} h_4 \setminus \operatorname{dom} h_2$. Then we arrive at $\mathit{refineheap}(pc_3, h_3, h_1)$.

$\square$

Lemma 12. *$s'_i \prec_{\mathcal{S}} s_i$ for all $1 \leq i \leq n+1$.*

Proof. Since $s_{n+1} = s'_{n+1}$, we get $s'_n \prec_{\mathcal{S}} s_n$ by Lemma 10. Then by induction: going backward with Lemma 11 for inductive step. □

Lemma 13. *$s'_1 \longrightarrow_{\mathcal{S}} s'_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s'_{n+1}$.*

Proof. It suffices to show for all $1 \leq i \leq n$, $s'_i \longrightarrow_{\mathcal{S}} s'_{i+1}$. Since $\left\langle s_i, s'_{i+1} \right\rangle \Rightarrow_{\mathcal{S}}^{-1} s'_i$, we have $s'_i \longrightarrow_{\mathcal{S}} s'_{i+1}$ by Lemma 8. □

Finally the proof of main theorem, Proposition 13:

Proof. By Lemma 12, we know the transition rule, $s'_i \longrightarrow_{\mathcal{S}} s'_{i+1}$, is not one of the lazy initialization rules. Then by Lemma 13 and Lemma 9, we get $c_i = \textit{default-concr}(T, E, s'_i) \longrightarrow_{C} \textit{default-concr}(T, E, s_{i+1}) = c_{i+1}$ for all $1 \leq i \leq n$. Since $c_i \in \gamma_s(s'_i) \subseteq \gamma_s(s_i)$, we have $c_i \in \gamma_s(s_i)$ for all $1 \leq i \leq n+1$. □

Using the soundness of SEL, we know that each path is covered by some symbolic trace. Since for each symbolic trace, we generate a concrete trace which covers the path, we have achieved complete path coverage.

7.2 Input Generation Formalization for Symbolic Execution with Lazier Initialization

7.2.1 Backtracking Rules for Symbolic Execution with Lazier Initialization

The backtracking rule FOO-A-BACK for rule FOO is defined as $\Sigma_{\mathcal{A}} \times \Sigma_{\mathcal{A}} \Rightarrow_{\mathcal{A}}^{-1} \Sigma_{\mathcal{A}}$. We modify the standard lazier backtracking rules as follows: if a symbolic location is initialized, the initialization is kept in the return state. For example, the IF_ACMPEQ2-A and IF_ACMPEQ3-A rules have the same backtracking rule IF_ACMPEQ-A-BACK. The GETFIELD1-A-BACK is the backtracking rule for the GETFIELD1-A rule. The backtrack rule for rule GETFIELD2-A is GETFIELD-S-BACK. The other backtracking rules are the same as the backtracking rules in the symbolic execution with lazy initialization shown in Section 7.1.1.

We use the bindings, $\sigma = (g, pc, l, \omega, h, \phi)$ and $\sigma' = (g', pc', l', \omega', h', \phi')$, for all the backtracking rules.

$$\text{IF\_ACMPEQ-A-BACK}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc_1 \qquad pc = pc'}{\langle \sigma, \sigma' \rangle \Rightarrow_{\mathcal{A}}^{-1} \sigma'}$$

$$\text{GETFIELD1-A-BACK}\ \frac{code(pc) = \texttt{getfield}\ f_\tau \qquad pc = pc'}{\langle \sigma, \sigma' \rangle \Rightarrow_{\mathcal{A}}^{-1} \sigma'}$$

Since each $\longrightarrow_{\mathcal{A}}$ transition may consist of multiple $\Rightarrow_a$ transitions, the backtracking of $\longrightarrow_{\mathcal{A}}$ will start from the last one and proceed backward. Suppose that $\sigma \longrightarrow_{\mathcal{A}} \sigma'$ consists of $n+1$ transitions $\sigma \Rightarrow_{\mathcal{A}} \sigma_1, \sigma_2 \Rightarrow_{\mathcal{A}} \sigma_2, \ldots,$ and $\sigma_n \Rightarrow_{\mathcal{A}} \sigma'$ in $\Rightarrow_a$. We know that the first n transitions just initialize symbolic locations and only the last transition does the real computation. By the backtracking rules of lazier symbolic execution, each initialization of symbolic location will backtrack to the second input state ($\langle a, a' \rangle \Rightarrow_{\mathcal{A}}^{-1} a'$). Thus the net effect of backtracking $\sigma \longrightarrow_{\mathcal{A}} \sigma'$ from σ' is the same as the just backtracking the last $\Rightarrow_{\mathcal{A}}$ rule from σ', that is, $\langle \sigma_n, \sigma' \rangle \Rightarrow_{\mathcal{A}}^{-1} \sigma''$.

Lemma 14. *For any $a_1 \longrightarrow_{\mathcal{A}} a_2$, after backtracking from a_3 we get an input state a_1' (by possible several backtracks). Then $a_1' \longrightarrow_{\mathcal{A}} a_3$ must hold. Further, $a_1' \longrightarrow_{\mathcal{A}} a_3$ and the transition rule is not one of those involving lazy/lazier initialization*[2].

Proof. We proceed by the rule induction on the transition of $\longrightarrow_{\mathcal{A}}$. If the last $\Rightarrow_{\mathcal{A}}$ transition in $a_1 \longrightarrow_{\mathcal{A}} a_2$ is $t \Rightarrow_{\mathcal{S}} a_2$ for some state t. Then we know that $\langle t, a_2 \rangle \Rightarrow_{\mathcal{S}}^{-1} a_1'$. Use Lemma 7, we get $a_1' \Rightarrow_{\mathcal{S}} a_2$ and the transition rule is not one of those involving lazy initialization. Otherwise, the last transition rule can only be IFNULL-A, IFNONNULL-A, or GETFIELD2-A. It is easy to show that the conclusion holds. Thus $a_1' \longrightarrow_{\mathcal{A}} a_2$ holds and the transition does not involve lazier/lazy initialization. □

7.2.2 Default Concretization Function for Symbolic Execution with Lazier Initialization

We now define a default concretization function $\textit{default-sym} : \Pi \times \Sigma_{\mathcal{A}} \to \Sigma_{\mathcal{S}}$ as $\textit{default-sym}(F, \sigma_a) = \sigma_s$ with $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$ for some F satisfies $\textit{default-sym-map}(F, \sigma_a)$. And $\textit{default-sym-map}(F, \sigma_a)$

[2] Lazy initialization refers to `getfield` and the field is not defined. Lazier initialization refers to initialize symbolic location to location.

if and only if

- $\forall \delta \in \textit{collect-sym-locs}(\sigma_a).F(\delta) \notin \operatorname{dom} h_a$;
- $\forall \delta_1, \delta_2 \in \textit{collect-sym-locs}(\sigma_a).F(\delta_1) = F(\delta_2) \implies \delta_1 = \delta_2$.

The idea is to create a new symbolic object for each symbolic location and add constraints to the path condition accordingly. It is obvious that $\textit{default-sym}(F, \sigma_a) \in \gamma_a(\sigma_a)$.

Lemma 15. *Suppose $a_1 \longrightarrow_{\mathcal{A}} a_2$ and the transition rule does not involve lazy/lazier initialization. Then $\textit{default-sym}(F, a_1) \longrightarrow_{\mathcal{S}} \textit{default-sym}(F, a_2)$ for some F such that $\textit{default-sym-map}(F, a_2)$ and the transition does not involve lazy initialization.*

Proof. We proceed with the rule induction on $\longrightarrow_{Laziest}$. □

7.2.3 Input Generation Algorithm for Symbolic Execution with Lazier Initialization and Proof

Given any trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots a_n \longrightarrow_{\mathcal{A}} a_{n+1}$. Suppose $t_i \Rightarrow_{\mathcal{A}} a_{i+1}$ is the last $\Rightarrow_{\mathcal{A}}$ transition in $a_i \longrightarrow_{\mathcal{A}} a_{i+1}$ for all $1 \leq i \leq n$. Define a sequence of states a'_i for $1 \leq i \leq n+1$ as

$$\begin{aligned}
&a'_{n+1} = a_{n+1} \\
&\langle t_n, a'_{n+1} \rangle \Rightarrow_{\mathcal{A}}^{-1} a'_n, \\
&\vdots \\
&\langle t_2, a'_3 \rangle \Rightarrow_{\mathcal{A}}^{-1} a'_2, \\
&\langle t_1, a'_2 \rangle \Rightarrow_{\mathcal{A}}^{-1} a'_1.
\end{aligned}$$

The applicability of backtracking rules is shown by Lemma 17. Then we apply the default concretization function for a'_i, where $1 \leq i \leq n+1$ for some F which satisfies $\textit{default-sym-map}(F, a_{n+1})$. And we get $s_1 = \textit{default-sym}(F, a'_1), \ldots, s_{n+1} = \textit{default-sym}(F, a'_{n+1})$.

Proposition 14. *1. $s_1 \longrightarrow_{\mathcal{S}} s_2 \cdots \longrightarrow_{\mathcal{S}} s_{n+1}$ and $s_i \in \gamma_a(a_i)$ for all $1 \leq i \leq n+1$;*
2. $\textit{default-concr}(T, E, s_1) \longrightarrow_{C} \textit{default-concr}(T, E, s_2) \cdots \longrightarrow_{C} \textit{default-concr}(T, E, s_{n+1})$ for some T, E satisfy the path condition of s_{n+1}.

To prove this main theorem, we need an additional definition and some lemmas.

Definition 9. *Relation $\prec_{\mathcal{A}}$: $\Sigma_{\mathcal{A}} \times \Sigma_{\mathcal{A}}$ as $a_1 \prec_{\mathcal{A}} a_2$ iff a_1 is similar to a_2 except that there are some fields of symbolic objects/arrays in a_1 are defined but not in a_2 and some of symbolic locations are initialized in a_1 but not in a_2.*

Precisely, $a_1 \prec_{\mathcal{A}} a_2$ for $a_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1)$ and $a_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2)$ if and only if following conditions hold

1. There exists some sequence of $(\delta_1, i_1), \ldots, (\delta_m, i_m)$ for some $m \geq 0$ where each i_i is a legal value for δ_i regarding to a_2 and let $t_1 = \textit{init-sym-loc}(a_2, \delta_1, i_1)$, $t_2 = \textit{init-sym-loc}(t_1, \delta_2, i_2)$, $\ldots$, $t_m = \textit{init-sym-loc}(t_{m-1}, \delta_m, i_m)$. Suppose $t_m = (g_{tm}, pc_1, l_{tm}, \omega_{tm}, h_{tm}, \phi_{tm})$, we have $g_{tm} = g_1, pc_1 = pc_2, l_1 = l_{tm}, \omega_1 = \omega_{tm}, \phi_{tm} \subseteq \phi_1$, and $\textit{refineheap}(pc_1, h_{tm}, h_1)$.

2. If $\textit{code}(pc_1)$ accesses heap, then the operands (in top of stack) a_1 can not be symbolic locations. For example, if $\textit{code}(pc_2) = \texttt{getfield}\,f$ then $\omega_1 = i :: \omega_1'$ for some $i \in \mathbf{Locs}$.

It is clear that $\gamma_a(a_1) \subseteq \gamma_a(a_2)$.

Lemma 16. *Suppose we have*

1. $\sigma_4 \prec_{\mathcal{A}} \sigma_2$ or

2. $\sigma_4 = \sigma_2$

and $\sigma_1 \longrightarrow_{\mathcal{A}} \sigma_2$. Let σ_3 be the outcome of backtracking from σ_4. Then $\sigma_3 \prec_{\mathcal{A}} \sigma_1$.

Proof. WLOG, we assume that $\sigma_4 \prec_{\mathcal{A}} \sigma_2$. We will prove by the rule induction of lazier semantics transitions: $\longrightarrow_{\mathcal{A}}$. We will use bindings: $\sigma_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1), \sigma_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2), \sigma_3 = (g_3, pc_3, l_3, \omega_3, h_3, \phi_3)$, and $\sigma_4 = (g_4, pc_4, l_4, \omega_4, h_4, \phi_4)$.

- Rule IADD: Since $\sigma_1 \Rightarrow_{\mathcal{A}} \sigma_2$ by rule IADD-S, we know $\omega_1 = v_1 :: v_2 :: \omega_1'$, $\textit{code}(pc_1) = \texttt{iadd}$, $\omega_2 = v' :: \omega_1'$, $pc_2 = \textit{next}(pc_1)$, $g_1 = g_2$, $l_1 = l_2$, $h_1 = h_2$, and $\phi_1 \subseteq \phi_2$. Since $\sigma_4 \prec_{\mathcal{A}} \sigma_2$ and the top element of ω_4 is a type of integer, we get $\sigma_3 \prec_{\mathcal{A}} \sigma_1$.

- Rule IF_ACMPEQ: WLOG, assume that $\sigma_1 \Rightarrow_{\mathcal{A}} t_1$ by rule IF_ACMEQ2-A; then $t_1 \Rightarrow_{\mathcal{A}} t_2$ by rule IF_ACMEQ3-A; finally $t_2 \Rightarrow_{\mathcal{A}} \sigma_2$ by rule IF_ACMEQ1-S. Then $\omega_1 = \delta' :: \delta'' :: \omega'_1$ for some δ', δ'', and ω'_1. Suppose the stack of t_2 is $\omega_{t_2} = i :: j :: \omega'_{t_2}$. Since $\sigma_4 \prec_{\mathcal{A}} \sigma_2$, there exists some sequence of $\overline{(\delta_i, i_i)}$ satisfies condition 1. We can just append (δ', i) and (δ'', j) in front of the sequence $\overline{(\delta_i, i_i)}$ for σ_3 and σ_1. By the backtracking rules IF_ACMEQ-A-BACK and IF_ACMPEQ-S-F-BACK, we get $\sigma_3 \prec_{\mathcal{A}} \sigma_1$.

- Rule GETFIELD f: WLOG, assume that $\sigma_1 \Rightarrow_{\mathcal{A}} t$ by rule GETFIELD1-A and $t \Rightarrow_{\mathcal{A}} \sigma_2$ by rule GETFIELD2-S. Then $\omega_1 = \delta :: \omega'_1$ for some δ and ω'_1 and the stack of t, $\omega_t = i :: \omega'_t$. Since $\sigma_4 \prec_{\mathcal{A}} \sigma_2$, there exists some sequence $\overline{(\delta_i, i_i)}$ satisfies condition 1. We can just append (δ, i) in front of sequence $\overline{(\delta_i, i_i)}$ for σ_3 and σ_1. By the backtracking rules GETFIELD1-A-BACK and GETFIELD-S-BACK, we get $\sigma_3 \prec_{\mathcal{A}} \sigma_1$.

$\square$

Lemma 17. *$a'_i \prec_{\mathcal{A}} a_i$ for all $1 \le i \le n+1$.*

Proof. Since $a_{n+1} = a'_{n+1}$, we have $a'_n \prec_{\mathcal{A}} a_n$ by case (2) of Lemma 16. Then by induction: going backward with case (1) of Lemma 16 for inductive step. $\square$

Lemma 18. *$a'_1 \longrightarrow_{\mathcal{A}} a'_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a'_{n+1}$. Further, all the rules are $\Rightarrow_{\mathcal{S}}$ and not involving lazy initialization.*

Proof. It suffices to show for all $1 \le i \le n$, $a'_i \longrightarrow_{\mathcal{A}} a'_{i+1}$. Since $a'_i \prec_{\mathcal{A}} a_i$, $a'_{i+1} \prec_{\mathcal{A}} a_{i+1}$, and $a_i \longrightarrow_{\mathcal{A}} a_{i+1}$, we have $a'_i \longrightarrow_{\mathcal{A}} a'_{i+1}$. By the definition of $\prec_{\mathcal{A}}$, we know that all the transitions rules are $\Rightarrow_{\mathcal{S}}$ and not involving lazy initialization. $\square$

Finally the proof of main theorem, Proposition 14:

Proof. By Lemma 18 and Lemma 15, we get $s_i = \textit{default-sym}(F, a'_i) \longrightarrow_{\mathcal{S}} \textit{default-sym}(F, a'_{i+1}) = s_{i+1}$ for all $1 \le i \le n$. Since $s_i \in \gamma_a(a'_i) \subseteq \gamma_a(a_i)$, we get $s_i \in \gamma_a(a_i)$ for all $1 \le i \le n+1$.

Since $a'_i \Rightarrow_{\mathcal{A}} a'_{i+1}$ does not involve lazy initialization, $s_i \Rightarrow_{\mathcal{S}} s_{i+1}$ does not involve lazy initialization. By Lemma 9, we have $\textit{default-concr}(T, E, s_1) \longrightarrow_C \textit{default-concr}(T, E, s_2) \cdots \longrightarrow_C \textit{default-concr}(T, E, s_{n+1})$ for some T, E satisfying the path condition of s_{n+1}. □

Since $\textit{default-concr}(T, E, s_i) \in \gamma_s(s_i)$ and $s_i \in \gamma_a(a_i)$, $\textit{default-concr}(T, E, s_i) \in \bigcup_{s \in \gamma_a(a)} \gamma_s(s)$. Therefore, for any lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \cdots \longrightarrow_{\mathcal{A}} a_n$, we generate a corresponding concrete trace $c_1 \longrightarrow_C c_2 \cdots \longrightarrow_C c_n$ and $c_i \in \gamma_s(\gamma_a(a_i))$ for all $1 \leq i \leq n$. Using the soundness result of SELA, we get complete path coverage.

7.3 Input Generation Formalization for Symbolic Execution with Lazier# Initialization

7.3.1 Backtracking Rules for Symbolic Execution with Laziest# Initialization

The backtracking rule FOO-B-BACK for rule FOO is defined as $\Sigma_{\mathcal{B}} \times \Sigma_{\mathcal{B}} \Rightarrow_{\mathcal{B}}^{-1} \Sigma_{\mathcal{B}}$. We modify the standard lazier# backtracking rules as follows: if a symbolic reference is initialized, the initialization is kept in the return state. For example, the IF_ACMPEQ2-B and IF_ACMPEQ3-B rules have the same backtracking rule IF_ACMPEQ-A-BACK. The GETFIELD1-A-BACK is the backtracking rule for the GETFIELD1-B rule. The backtracking rule for rule GETFIELD2-B is GETFIELD-S-BACK. The other backtracking rules are the same as the backtracking rules in the symbolic execution with lazy/lazier# initialization shown in Section 7.1.1 and Section 7.3.1.

Since each $\longrightarrow_{\mathcal{B}}$ transition may consist of multiple $\Rightarrow_{\mathcal{B}}$ transitions, the backtracking of $\longrightarrow_{\mathcal{B}}$ will start from the last one and proceed backward. Suppose that $\sigma \longrightarrow_{\mathcal{B}} \sigma'$ consists of $n + 1$ transitions $\sigma \Rightarrow_{\mathcal{B}} \sigma_1, \sigma_2 \Rightarrow_{\mathcal{B}} \sigma_2, \ldots,$ and $\sigma_n \Rightarrow_{\mathcal{B}} \sigma'$ in $\Rightarrow_b$. We know that the first n transitions just initialize symbolic locations/references and only the last transition does the real computation. By the backtracking rules of lazier# symbolic execution, each initialization of symbolic location/reference will backtrack to the second input state ($\langle b, b' \rangle \Rightarrow_{\mathcal{B}}^{-1} b'$). Thus the net effect of backtracking $\sigma \longrightarrow_{\mathcal{B}} \sigma'$ from σ' is the same as the just backtracking the last $\Rightarrow_{\mathcal{B}}$ rule from σ', that is, $\langle \sigma_n, \sigma' \rangle \Rightarrow_{\mathcal{B}}^{-1} \sigma''$.

Lemma 19. *For any $b_1 \longrightarrow_{\mathcal{B}} b_2$, after backtracking from b_2, we get an input state b_1' (possible by several backtracks). Then $b_1' \Rightarrow_{\mathcal{B}} b_2$ must hold. Further, the transition rule is not one of those involving lazy initialization.*

Proof. Suppose the last $\Rightarrow_{\mathcal{B}}$ transition in $b_1 \longrightarrow_{\mathcal{B}} b_2$ is $t \Rightarrow_{\mathcal{B}} b_2$ for some state t. Then we know that $\langle t, b_2\rangle \Rightarrow_{\mathcal{B}}^{-1} b_1'$. We can use the rule induction on the instruction and get $b_1' \Rightarrow_{\mathcal{S}} b_2$ and the transition rule is not one of those involving lazy initialization. Thus $b_1' \longrightarrow_{\mathcal{B}} b_2$ holds. □

7.3.2 Default Concretization Function for Symbolic Execution with Lazier# Initialization

We now define a default concretization function *default-lazier* : $\Xi \times \Sigma_{\mathcal{B}} \rightarrow \Sigma_{\mathcal{A}}$ as *default-lazier*$(G, \sigma_b) = \sigma_a$ with $\sigma_a \in \mathcal{ST}_b[\![\sigma_b]\!](G)$ for some G satisfies *default-symref-map*(G, σ_b). And *default-symref-map*(G, σ_b) if and only if $\forall \hat{\delta} \in$ *collect-sym-refs*$(\sigma_b).G(\hat{\delta}) =$ NULL. The idea is to set each symbolic reference to NULL.

Lemma 20. *Suppose $b_1 \longrightarrow_{\mathcal{B}} b_2$ and $b_1 \Rightarrow_{\mathcal{B}} b_2$ the transition rule does not involve lazy initialization. Then default-lazier$(G, b_1) \longrightarrow_{\mathcal{S}}$ default-lazier(G, b_2) for some G such that default-symref-map(G, b_2).*

Proof. Proof by the rule induction on $\longrightarrow_{\mathcal{B}}$. □

7.3.3 Input Generation Algorithm for Symbolic Execution with Lazier# Initialization and Proof

Given any trace $b_1 \longrightarrow_{\mathcal{B}} b_2 \longrightarrow_{\mathcal{B}} \cdots b_n \longrightarrow_{\mathcal{B}} b_{n+1}$. Suppose $t_i \Rightarrow_{\mathcal{A}} b_{i+1}$ is the last $\Rightarrow_{\mathcal{B}}$ transition in $b_i \longrightarrow_{\mathcal{B}} b_{i+1}$ for all $1 \leq i \leq n$. Define a sequence of states b_i' for $1 \leq i \leq n+1$ as

$$\begin{aligned}
&b_{n+1}' = b_{n+1}\\
&\langle t_n, b_{n+1}'\rangle \Rightarrow_{\mathcal{A}}^{-1} b_n',\\
&\vdots\\
&\langle t_2, b_3'\rangle \Rightarrow_{\mathcal{A}}^{-1} b_2',\\
&\langle t_1, b_2'\rangle \Rightarrow_{\mathcal{A}}^{-1} b_1'.
\end{aligned}$$

The applicability of backtracking rules is shown by Lemma 22. Then we apply the default concretization function for b'_i, where $1 \leq i \leq n{+}1$ for some G which satisfies *default-symref-map*(G, b_{n+1}). And we define $a_1 = \textit{default-lazier}(G, b'_1), \ldots, a_{n+1} = \textit{default-lazier}(G, b'_{n+1})$. Define $s_1 = \textit{default-sym}(F, a_1)$, $\ldots$, $s_{n+1} = \textit{default-sym}(F, a_{n+1})$ where F satisfies *default-sym-map*(F, a_{n+1}).

Proposition 15. *1.* $a_1 \longrightarrow_{\mathcal{A}} a_2 \cdots \longrightarrow_{\mathcal{A}} a_{n+1}$.
2. $s_1 \longrightarrow_{\mathcal{S}} s_2 \cdots \longrightarrow_{\mathcal{S}} s_{n+1}$ *and* $s_i \in \gamma_a(a_i)$ *for all* $1 \leq i \leq n+1$*;*
3. default-concr$(T, E, s_1) \longrightarrow_{C}$ *default-concr*$(T, E, s_2) \cdots \longrightarrow_{C}$ *default-concr*(T, E, s_{n+1}) *for some* T, E *satisfy the path condition of* s_{n+1}.

To prove this main theorem, we need an additional definition and some lemmas.

Definition 10. *Relation* $\prec_{\mathcal{B}}$: $\Sigma_{\mathcal{B}} \times \Sigma_{\mathcal{B}}$ *as* $b_1 \prec_{\mathcal{B}} b_2$ *iff* b_1 *is similar to* b_2 *except that there are some fields of symbolic objects/arrays in* b_1 *are defined but not in* b_2*; some of symbolic locations are initialized in* b_1 *but not in* b_2*; some of symbolic references are initialized in* b_1 *but not in* b_2*.*

Precisely, $b_1 \prec_{\mathcal{B}} b_2$ for $b_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1)$ and $b_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2)$ if and only if following conditions hold

1. There exists a subset of symbolic references, $psr \subseteq$ **SymRefs** and a state $t = \mathcal{ST}_b[\![b_2]\!](G\mid_{psr})$ such that $G \in \textit{legal-env}(b_2)$. Further, there exists some sequence of $(\delta_1, i_1), \ldots, (\delta_m, i_m)$ for some $m \geq 0$ where each i_i is a legal value for δ_i regarding to t and let $t_1 = \textit{init-sym-loc}(t, \delta_1, i_1)$, $t_2 = \textit{init-sym-loc}(t_1, \delta_2, i_2), \ldots, t_m = \textit{init-sym-loc}(t_{m-1}, \delta_m, i_m)$. Suppose $t_m = (g_{tm}, pc_1, l_{tm}, \omega_{tm}, h_{tm}, \phi_{tm})$, we have $g_{tm} = g_1, pc_1 = pc_2, l_1 = l_{tm}, \omega_1 = \omega_{tm}, \phi_{tm} \subseteq \phi_1$, and *refineheap*$(pc_1, h_{tm}, h_1)$.

2. If *code*(pc_1) accesses heap, then the operand (in top of stack) b_1 can not be a symbolic location or symbolic reference. For example, if $\textit{code}(pc_2) = \texttt{getfield}f$ then $\omega_1 = i :: \omega'_1$ for some $i \in$ **Locs**.

It is clear that $\gamma_b(b_1) \subseteq \gamma_b(a_2)$.

Lemma 21. *Suppose we have*

1. $\sigma_4 \prec_{\mathcal{B}} \sigma_2$ or

2. $\sigma_4 = \sigma_2$

and $\sigma_1 \longrightarrow_{\mathcal{B}} \sigma_2$. Let σ_3 be the outcome of backtracking from σ_4. Then $\sigma_3 \prec_{\mathcal{B}} \sigma_1$.

Proof. WLOG, we assume that $\sigma_4 \prec_{\mathcal{B}} \sigma_2$. We will prove by the rule induction on lazier# semantics transitions: $\longrightarrow_{\mathcal{B}}$. We will use bindings: $\sigma_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1), \sigma_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2), \sigma_3 = (g_3, pc_3, l_3, \omega_3, h_3, \phi_3)$, and $\sigma_4 = (g_4, pc_4, l_4, \omega_4, h_4, \phi_4)$.

- Rule IADD: Since $\sigma_1 \Rightarrow_{\mathcal{A}} \sigma_2$ by rule IADD-S, we know $\omega_1 = v_1 :: v_2 :: \omega_1'$, $code(pc_1) =$ `iadd`, $\omega_2 = v' :: \omega_1'$, $pc_2 = next(pc_1)$, $g_1 = g_2$, $l_1 = l_2$, $h_1 = h_2$, and $\phi_1 \subseteq \phi_2$. Since $\sigma_4 \prec_{\mathcal{B}} \sigma_2$ and the top element of ω_4 is a type of integer, we get $\sigma_3 \prec_{\mathcal{B}} \sigma_1$.

- Rule IF_ACMPEQ: WLOG, assume that $\sigma_1 \Rightarrow_{\mathcal{B}} t_1$ by rule IF_ACMEQ2-B; then $t_1 \Rightarrow_{\mathcal{B}} t_2$ by rule IF_ACMEQ3-B; then $t_3 \Rightarrow_{\mathcal{B}} t_4$ by rule IF_ACMEQ2-A; then $t_4 \Rightarrow_{\mathcal{B}} t_5$ by rule IF_ACMEQ3-A; finally $t_5 \Rightarrow_{\mathcal{B}} \sigma_2$ by rule IF_ACMEQ1-S. Then $\omega_1 = \hat{\delta}' :: \hat{\delta}'' :: \omega_1'$ for some $\hat{\delta}', \hat{\delta}''$, and ω_1'. Suppose the stack of t_3 is $\omega_{t_3} = \delta' :: \delta'' :: \omega_{t_3}'$ and the stack of t_5 is $\omega_{t_5} = i :: j :: \omega_{t_5}'$. Since $\sigma_4 \prec_{\mathcal{B}} \sigma_2$, there exist psr, G, and some sequence of $\overline{(\delta_i, i_i)}$ satisfy condition 1. We can just let $psr' = psr \cup \{\hat{\delta}', \hat{\delta}''\}$ and $G' = G[\hat{\delta}' \mapsto \delta'][\hat{\delta}'' \mapsto \delta'']$ and append (δ', i) and (δ'', j) in front of the sequence $\overline{(\delta_i, i_i)}$ for σ_3 and σ_1. By the backtracking rules IF_ACMEQ-A-BACK and IF_ACMPEQ-S-F-BACK, we get $\sigma_3 \prec_{\mathcal{B}} \sigma_1$.

- Rule GETFIELD f: WLOG, assume that $\sigma_1 \Rightarrow_{\mathcal{B}} t_1$ by rule GETFIELD1-B and $t_1 \Rightarrow_{\mathcal{B}} t_3$ and $t_2 \Rightarrow \sigma_2$ by rule GETFIELD2-S. Then $\omega_1 = \hat{\delta} :: \omega_1'$ for some $\hat{\delta}$ and ω_1' and the stack of t_1, $\omega_{t_1} = \delta :: \omega_{t_1}'$ and the stack of t_2, $\omega_{t_2} = i :: \omega_{t_2}'$. Since $\sigma_4 \prec_{\mathcal{B}} \sigma_2$, there exist psr, G, and some sequence $\overline{(\delta_i, i_i)}$ satisfy condition 1. We can let $psr' = psr \cup \{\hat{\delta}\}$, $G' = G[\hat{\delta} \mapsto \delta]$, and append (δ, i) in front of the sequence $\overline{(\delta_i, i_i)}$ for σ_3 and σ_1. By the backtracking rules GETFIELD1-A-BACK and GETFIELD-S-BACK, we get $\sigma_3 \prec_{\mathcal{B}} \sigma_1$.

□

Lemma 22. *$b'_i \prec_{\mathcal{B}} b_i$ for all $1 \leq i \leq n+1$.*

Proof. Since $b_{n+1} = b'_{n+1}$, we have $b'_n \prec_{\mathcal{B}} b_n$ by case (2) of Lemma 21. Then by induction: going backward with case (1) of Lemma 21 for inductive step. □

Lemma 23. *$b'_1 \longrightarrow_{\mathcal{B}} b'_2 \longrightarrow_{\mathcal{B}} \cdots \longrightarrow_{\mathcal{B}} b'_{n+1}$. Further, each transition consists of just one $\Rightarrow_{\mathcal{B}}$ and does not involve lazy initialization.*

Proof. It suffices to show for all $1 \leq i \leq n$, $b'_i \longrightarrow_{\mathcal{B}} b'_{i+1}$. Since $b'_i \prec_{\mathcal{B}} b_i$, $b'_{i+1} \prec_{\mathcal{B}} b_{i+1}$, and $b_i \longrightarrow_{\mathcal{B}} b_{i+1}$, we have $b'_i \longrightarrow_{\mathcal{B}} b'_{i+1}$. By the definition of $\prec_{\mathcal{B}}$, we know that all the transition rules are one step and do not involve lazy initialization. □

Finally the proof of main theorem, Proposition 15:

Proof. By Lemma 23 and Lemma 20, we get $a_i = \textit{default-lazier}(G, b'_i) \longrightarrow_{\mathcal{A}} \textit{default-lazier}(G, b'_{i+1}) = a_{i+1}$ for all $1 \leq i \leq n$. Further, $s_i = \textit{default-sym}(G, a_i) \longrightarrow_{\mathcal{S}} \textit{default-sym}(G, a_{i+1})$. Since $s_i \in \gamma_a(a_i)$ and $a_i \in \gamma_b(b'_i) \subseteq \gamma_b(b_i)$, we get $s_i \in \gamma_b(\gamma_a(a_i))$ for all $1 \leq i \leq n+1$. Since $b'_i \Rightarrow_{\mathcal{B}} b'_{i+1}$ does not involve lazy initialization, $s_i \Rightarrow_{\mathcal{S}} s_{i+1}$ does not involve lazy initialization. By Lemma 9, we have $\textit{default-concr}(T, E, s_1) \longrightarrow_{C} \textit{default-concr}(T, E, s_2) \cdots \longrightarrow_{C} \textit{default-concr}(T, E, s_{n+1})$ for some T, E satisfying the path condition of s_{n+1}. □

Since $\textit{default-concr}(T, E, s_i) \in \gamma_s(s_i)$ and $s_i \in \gamma_a(a_i)$, $\textit{default-concr}(T, E, s_i) \in \bigcup_{a \in \gamma_b(b_i)} \bigcup_{s \in \gamma_a(a)} \gamma_s(s)$. Therefore, for any lazier# symbolic trace $b_1 \longrightarrow_{\mathcal{B}} b_2 \cdots \longrightarrow_{\mathcal{B}} b_n$, we generate a corresponding concrete trace $c_1 \longrightarrow_{C} c_2 \cdots \longrightarrow_{C} c_n$ and $c_i \in \gamma_s(\gamma_a(\gamma_b(b_i)))$ for all $1 \leq i \leq n$. Using the soundness result of SELB, we get the complete path coverage.

Chapter 8

Examples

This chapter presents several examples to illustrate the Kiasan/KUnit approach. The examples include insertion sort, binary search tree, and red-black tree. For each example, we just exam one method: sort for insertion sort, insert for binary search tree, and remove for red-black tree. For each method, we present the source code and together with the Kiasan specification. Then we give the Kiasan result (numbers of pre/post-states) and KUnit result: pre/post-state graphs and JUnit test for one trace. Detailed data such as timing and number of states is presented in Chapter 9.

8.1 Insertion Sort

Insertion sort is a simple sort algorithm that has the time complexity of $O(n^2)$. The implementation of insertion sort is taken from a data structure textbook [63]. We will first present the integer version of insertion sort as shown in Figure 8.1. The method to be checked is `insertionSort` (lines 3-11). Lines 1-2 are the specification for `insertionSort`. Essentially, the specification states that the precondition of the method is that the input array `a` is non-null and the postcondition is that `a` is sorted. The postcondition is specified by a method `isSorted` (lines 12-19) which returns true if and only if the input array is ascending.

From this example, we can see that Kiasan specification uses Annotation from Java 1.5 and the precondition and postcondition are legal Java expressions which can include method calls. Kiasan requires that the method calls or expressions in the specification be observably pure, that is, no visible effect to the user observable states.

```
@Assertion(value = { @Case(pre = "a␣!=null",
        post = "isSorted(a)") })
public void insertionSort(int[ ] a ) {
    int j;
    for( int p = 1; p < a.length; p++ ) {
        int tmp = a[ p ];
        for( j = p; j > 0 && tmp<a[ j - 1 ]; j-- )
             a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
private  boolean isSorted(int[] a) {
    for(int i=0;i<a.length-1;i++) {
        if(a[i]>a[i+1]) {
            return false;
        }
    }
    return true;
}
```

Figure 8.1: *Insertion Sort Method (Integer) with Kiasan Specification*

Figure 8.2 shows the implementation of insertion sort for `Comparables`. In order to check this method, we need to specify that permuting the elements of the input array does not affect the results of comparisons between array elements. This is done by using regions. The named regions are initially disjointed memory areas. `Region.arrayReg(a,0,1)` at line 2 specifies that array `a` is in region 0 and its elements are in region 1. Region 1 does not have access to region 0 but region 0 can reach region 1. The comparisons are happened in region 1 and the updates are in region 0. So Kiasan can conclude the comparison results in the method body are preserved when evaluating the postcondition.

8.1.1 Results

Table 8.1 shows the Kiasan analysis result for insertion sort. k denotes the resource-bound. For all of the examples, we did not need to use loop bounding. **Pre**, **B-Post**, and **A-Post** denote the number of states at M's entry, before executing M's postcondition, and after executing M's postcondition respectively.

As can be expected, the number of **A-Post** follows the formula $\sum_{i=0}^{k} i!$ in the Comparable version. This holds because for any array with n elements, there are possible $n!$ orderings (permutations). In our k-bounded symbolic executions, the state-space for $k = i$ includes cases for $k < i$.

```
@Assertion(value =
  { @Case(pre = "Region.arrayReg(a,0,1)␣&&␣noNullElement(a)",
               post = "isSorted(a)") })
public static void insertionSort( Comparable [ ] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Comparable tmp = a[ p ];
        for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
              a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
private boolean noNullElement(Object[] a) {
    for(int i=0; i<a.length; i++) {
        if(a[i]==null)
              return false;
    }
    return true;
}

private  boolean isSorted(Comparable[] a) {
    for(int i=0;i<a.length-1;i++) {
    if(a[i].compareTo(a[i+1])>0) {
         return false;
    }
    return true;
 }
```

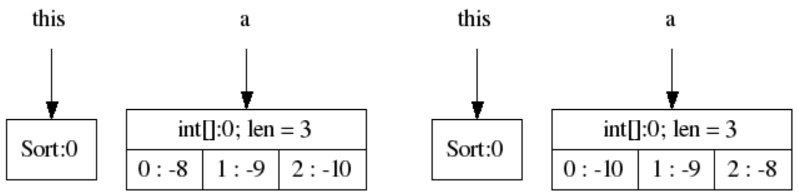
Figure 8.2: *Insertion Sort Method (Comparable) with Kiasan Specification*

Example	k	Pre	B-Post	A-Post
integer version	1	1	1	1
	2	1	3	3
	3	1	9	9
	4	1	33	33
Comparable Version	1	2	2	2
	2	3	4	4
	3	4	10	10
	4	5	34	34

Table 8.1: *Kiasan Result for Insertion Sort*

In the case for the integer version, the formula is $\sum_{i=1}^{k} i!$ because the cases of $k = 1$ and $k = 0$ are represented by one symbolic execution path. This happens because the precondition for the `Comparable` version requires that all elements of the array are non-null, thus, the precondition expands the array elements up to the bound. In the integer version, there is no constraint on the elements of the array in the precondition, thus, both arrays with zero and one element are not expanded into different execution paths. This also causes **Pre** numbers of the integer and Comparable version to be different.

As shown in Chapter 6, KUnit generates pre/post-state heap graphs and a JUnit test case for each trace. So for $k = 3$, KUnit generates total 9 test cases and pre/post-state graphs for the integer version. Figure 8.3 shows the pre and post-states for one trace and Figure 8.4 shows the corresponding JUnit code. Figure 8.5 shows the corresponding comparable version of JUnit test case code generated by KUnit. Furthermore, for $k = 2$, KUnit shows that the insertion sort code reaches 100% branch coverage.



(a) pre-state (b) post-state

Figure 8.3: *KUnit Output for Insertion Sort (Excerpt)*

8.2 Binary Search Tree

Binary search tree is a special kind of binary tree with an additional key in each node and those keys have following ordering properties:

- keys are from a total ordered set;
- all the keys in the left subtree of a node are less than the node's key;

```
import junit.framework.*;
import java.lang.reflect.Field;
public class InsertionSortTest8 extends TestCase{
    static int INDEX=0;
    public static junit.framework.Test suite() {
        return new TestSuite(InsertionSortTest8.class);}

    public void testInsertionSort() throws Exception{
        Class clazzintelement$Sort = Class.forName(”intelement.Sort”);
        intelement.Sort this_;
        intelement.Sort sort_0=null;
        sort_0 = (intelement.Sort)clazzintelement$Sort.newInstance();
        this_=sort_0;
        int[] a;
        int[] int___0=null;
        int___0=new int[3];
        int___0[2]=-10;
        int___0[1]=-9;
        int___0[0]=-8;
        a=int___0;
        this_.insertionSort(a);
        assertEquals(this_,sort_0);
        assertEquals(a,int___0);
        assertEquals(int___0[2],-8);
        assertEquals(int___0[1],-9);
        assertEquals(int___0[0],-10);
        assertEquals(int___0.length,3);
    }
}
```

Figure 8.4: *KUnit Generated JUnit Test Case for One Trace*

```
import junit.framework.*;
import java.lang.reflect.Field;
public class InsertionSortTest9 extends TestCase {
    static int INDEX = 5;
    public static junit.framework.Test suite() {
        return new TestSuite(InsertionSortTest9.class); }
    static int interfaceCall_1(Comparable a0,Object a1){return -10;}
    static int interfaceCall_2(Comparable a0,Object a1){return -10;}
    static int interfaceCall_3(Comparable a0,Object a1){return -10;}
    static int interfaceCall_0(Comparable a0,Object a1){return -10;}
    static int interfaceCall_4(Comparable a0,Object a1){return -10;}
    public void testInsertionSort() throws Exception {
        Class clazzcomparable$Sort = Class.forName(”comparable.Sort”);
        Class clazzjava$lang$Comparable = Class.forName(”Comparable”);
        java.lang.Comparable comparable_2 = new java.lang.Comparable(){
            public int compareTo(java.lang.Object arg1) {
                INDEX--;
                switch (INDEX) {
                case 2: return interfaceCall_2(this, arg1);
                case 1: return interfaceCall_1(this, arg1);
                case 3: return interfaceCall_3(this, arg1);
                default: throw new Error(); }
            }
        };
        java.lang.Comparable comparable_1 =new java.lang.Comparable(){
            public int compareTo(java.lang.Object arg1) {
                INDEX--;
                switch (INDEX) {
                case 4: return interfaceCall_4(this, arg1);
                case 0: return interfaceCall_0(this, arg1);
                default: throw new Error();
                }
            }
        };
        comparable.Sort this_,sort_0 = null;
        this_=sort_0=(comparable.Sort)clazzcomparable$Sort.newInstance();
        java.lang.Comparable[] a,comparable___0 = new Comparable[3];
        comparable___0[2] = comparable_2;
        comparable___0[1] = comparable_1;
        comparable_0=(Comparable)clazzjava$lang$Comparable.newInstance();
        comparable___0[0] = comparable_0;
        a = comparable___0;
        this_.insertionSort(a);
        assertEquals(this_, sort_0);
        assertEquals(a, comparable___0);
        assertEquals(comparable___0[2], comparable_0);
        assertEquals(comparable___0[1], comparable_1);
        assertEquals(comparable___0[0], comparable_2);
        assertEquals(comparable___0.length, 3);
    }
}
```

Figure 8.5: *KUnit Generated JUnit Test Case for One Trace(Comparable)*

```
boolean repOK(BinaryNode t) {
    return repOK(t,new Range());
}
boolean repOK(BinaryNode t, Range range) {
    if(t == null) {
        return true;
    }
    if(!range.inRange(t.element)) {
        return false;
    }
    boolean ret=true;
    ret=repOK(t.left,range.setUpper(t.element));
    ret=ret &&
       repOK(t.right,range.setLower(t.element));
    return ret;
}
```

Figure 8.6: *Binary Search Tree Invariant Specification*

```
@Assertion(value = { @Case(pre = "(repOK(root))",
                           post = "repOK(root)") })
public void insert( int x ) {
    root = insert( x, root );
}
private BinaryNode insert( int x, BinaryNode t ) {
    if( t == null )
        t = new BinaryNode( x, null, null );
    else if( x < t.element)
        t.left = insert( x, t.left );
    else if( x> t.element )
        t.right = insert( x, t.right );
    else
        ;  // Duplicate; do nothing
    return t;
}
```

Figure 8.7: *Binary Search Tree Insert Method*

- all the keys in the right subtree of a node are greater than the node's key.

We take a binary search tree implementation from [63]. Figure 8.6 shows the invariant of binary search tree. For each node, there is a range (an upper bound and lower bound) that associates with the node. The `repOK` checks recursively that the element in the node is in the range. Thus the invariant basically checks the ordering of the keys in the tree. Furthermore, this ordering implies that the structure must be acyclic, otherwise the ordering property is violated.

Figure 8.7 shows the `insert` method of binary search tree. The specification is just the invariant.

Figure 8.8 shows part of `insert` method for binary search trees with object elements and a `Comparator` for enforcing the total order. In addition to the invariant, the specification contains region specification to facilitate the checking of the invariant. The region specification shows that there are two user specified regions: 0 and 1. The `root` is in region 0 and parameter x is in region 1. The field accesses of `BinaryNode` from region 0 through fields `left,right` end up in region 0 but accesses through field `element` arrive in region 1. In short, the tree structure is in region 0 and the tree elements and the element to be inserted are in region 1.

```
@Assertion(value = { @Case(
pre = "Region.setObjReg(this.root,0)␣&&␣Region.setObjReg(x,1)␣&&
Region.setFldReg(\"comparator.BinaryNode\",\"left\",0,0)␣&&
Region.setFldReg(\"comparator.BinaryNode\",\"right\",0,0)␣&&
Region.setFldReg(\"comparator.BinaryNode\",\"element\",0,1)␣&&
this.comparator␣!=␣null␣␣&&␣(repOK(root))",
post = "repOK(root)") })
public void insert(@NonNull T x ) {
   root = insert( x, root );
}
```

Figure 8.8: *Binary Search Tree Insert Method (Comparator)*

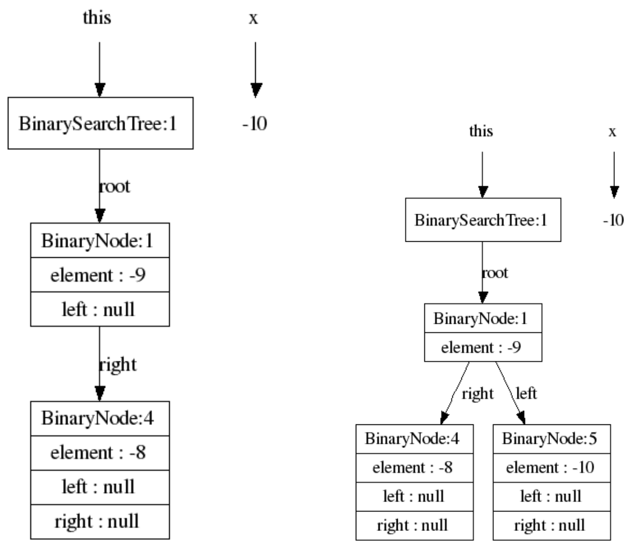
8.2.1 Results

Table 8.2 shows the Kiasan result for the `insert` method of binary search tree. All the machine settings and table arrangements are the same as the insertion sort example. For any k, we are checking essentially all the non-isomorphic binary search trees with height less than k since the wrapper object has bound k and the root has the bound $k-1$. For each k, the number of pre-states equals to the number of non-isomorphic binary search trees with height less than k. The number of post-state equals to $(BST_n \times \mathbb{Z})/R'$ as shown in Section 4.2. The numbers of pre/post-states are optimal in the sense that they match the minimal numbers of cases of theoretical calculation shown in Section 4.2.

Figure 8.9 shows the pre and post-states for one path with $k = 2$. Figures 8.10 and 8.11 show the generated JUnit code for the path that corresponds to 8.9. The generated object version JUnit test case is shown in Figures 8.12 and 8.13. Furthermore, that for $k = 2$, the `insert` method of binary search tree reaches 100% branch coverage.

Example	*k*	**Pre**	**B-Post**	**A-Post**
	1	2	4	4
`insert`	2	5	21	21
`(int)`	3	26	236	236
	1	2	4	4
`insert`	2	5	21	21
`(Comparator)`	3	26	236	236

Table 8.2: *Kiasan Results for Binary Search Insert Method*



(a) pre-state (b) post-state

Figure 8.9: *KUnit Output for Binary Search Tree Insert (Excerpt)*

```
import junit.framework.*;
import java.lang.reflect.Field;
public class InsertTest4 extends TestCase {
   static int INDEX = 0;
   public static junit.framework.Test suite() {
      return new TestSuite(InsertTest4.class); }
   public void testInsert() throws Exception {
      Class clazzintkey$BinaryNode = Class.forName("intkey.BinaryNode");
      Class clazzintkey$BinarySearchTree = Class.forName("intkey.BinarySearchTree");
      intkey.BinarySearchTree this_;
      intkey.BinarySearchTree binarySearchTree_0 = null;
      binarySearchTree_0 = (intkey.BinarySearchTree) clazzintkey$BinarySearchTree.newInstance();
      intkey.BinaryNode binaryNode_1 binaryNode_1 = (intkey.BinaryNode) clazzintkey$BinaryNode.
            newInstance();
      intkey.BinaryNode binaryNode_4 = (intkey.BinaryNode) clazzintkey$BinaryNode.newInstance();
      // binaryNode_4.right=null;
      for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("right")) {
            f.setAccessible(true);
            f.set(binaryNode_4, null);
            break;
         }
      }
      // binaryNode_4.left=null;
      for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("left")) {
            f.setAccessible(true);
            f.set(binaryNode_4, null);
            break;
         }
      }
      // binaryNode_4.element=-8;
      for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("element")) {
            f.setAccessible(true);
            f.setInt(binaryNode_4, -8);
            break;
         }
      }
      // binaryNode_1.right=binaryNode_4;
      for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("right")) {
            f.setAccessible(true);
            f.set(binaryNode_1, binaryNode_4);
            break;
         }
      }
      // binaryNode_1.left=null;
      for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("left")) {
            f.setAccessible(true);
            f.set(binaryNode_1, null);
            break;
         }
      }
      // binaryNode_1.element=-9;
      for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("element")) {
            f.setAccessible(true);
            f.setInt(binaryNode_1, -9);
            break;
         } }
```

Figure 8.10: *KUnit Generated JUnit Test Case for Insert*

```
    // binarySearchTree_0.root=binaryNode_1;
    for (Field f : clazzintkey$BinarySearchTree.getDeclaredFields()) {
      if (f.getName().equals("root")) {
        f.setAccessible(true);
        f.set(binarySearchTree_0, binaryNode_1);
        break; } }
    this_ = binarySearchTree_0; int x = -10; this_.insert(x);
    assertEquals(this_, binarySearchTree_0);
    // binarySearchTree_0.root==binaryNode_1;
    for (Field f : clazzintkey$BinarySearchTree.getDeclaredFields()) {
      if (f.getName().equals("root")) {
        f.setAccessible(true);
        assertEquals(binaryNode_1, f.get(binarySearchTree_0)); break; } }
    // binaryNode_1.right==binaryNode_4;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("right")) {
        f.setAccessible(true);
        assertEquals(binaryNode_4, f.get(binaryNode_1)); break; } }
    // binaryNode_4.right==null;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("right")) {
        f.setAccessible(true);
        assertEquals(null, f.get(binaryNode_4)); break; } }
    // binaryNode_4.left==null;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("left")) {
        f.setAccessible(true);
        assertEquals(null, f.get(binaryNode_4)); break; } }
    // binaryNode_4.element==-8;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("element")) {
        f.setAccessible(true);
        assertEquals(-8, f.getInt(binaryNode_4)); break; } }
    intkey.BinaryNode binaryNode_5 = null;
    // binaryNode_5=binaryNode_1.left;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("left")) {
        f.setAccessible(true);
        binaryNode_5 = (intkey.BinaryNode) f.get(binaryNode_1); break; } }
    // binaryNode_5.element==-10;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("element")) {
        f.setAccessible(true);
        assertEquals(-10, f.getInt(binaryNode_5)); break; } }
    // binaryNode_5.left==null;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("left")) {
        f.setAccessible(true);
        assertEquals(null, f.get(binaryNode_5)); break; } }
    // binaryNode_5.right==null;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("right")) {
        f.setAccessible(true);
        assertEquals(null, f.get(binaryNode_5)); break; } }
    // binaryNode_1.element==-9;
    for (Field f : clazzintkey$BinaryNode.getDeclaredFields()) {
      if (f.getName().equals("element")) {
        f.setAccessible(true);
        assertEquals(-9, f.getInt(binaryNode_1)); break; } }
    assertEquals(x, -10); } }
```

Figure 8.11: *KUnit Generated JUnit Test Case for Insert (Continue)*

```
import junit.framework.*;
import java.lang.reflect.Field;
public class InsertTest4 extends TestCase { static int INDEX = 4;
   public static junit.framework.Test suite() {
      return new TestSuite(InsertTest4.class); }
   static int interfaceCall_0(java.util.Comparator arg0,Object arg1,Object arg2) { return 1; }
   static int interfaceCall_1(java.util.Comparator arg0,Object arg1,Object arg2) { return -10; }
   static int interfaceCall_2(java.util.Comparator arg0,Object arg1,Object arg2) { return -10; }
   static int interfaceCall_3(java.util.Comparator arg0,Object arg1,Object arg2) { return 1; }
   public void testInsert() throws Exception {
      Class clazzjava$lang$Object = Class.forName("java.lang.Object");
      Class clazzjava$util$Comparator = Class.forName("java.util.Comparator");
      Class clazzcomparator$BinaryNode = Class.forName("comparator.BinaryNode");
      Class clazzcomparator$BinarySearchTree = Class.forName("comparator.BinarySearchTree");
      java.util.Comparator comparator_0 = new java.util.Comparator() {
         public int compare(java.lang.Object arg1, java.lang.Object arg2) {
            INDEX--;
            switch (INDEX) {
            case 2:
               return interfaceCall_2(this, arg1, arg2);
            case 1:
               return interfaceCall_1(this, arg1, arg2);
            case 3:
               return interfaceCall_3(this, arg1, arg2);
            case 0:
               return interfaceCall_0(this, arg1, arg2);
            default:
               throw new Error(); } } };
      comparator.BinarySearchTree this_;
      comparator.BinarySearchTree binarySearchTree_0 = null;
      binarySearchTree_0 = (comparator.BinarySearchTree) clazzcomparator$BinarySearchTree.newInstance();
      comparator.BinaryNode binaryNode_1 = null;
      binaryNode_1 = (comparator.BinaryNode) clazzcomparator$BinaryNode
            .newInstance();
      comparator.BinaryNode binaryNode_4 = null;
      binaryNode_4 = (comparator.BinaryNode) clazzcomparator$BinaryNode
            .newInstance();
      // binaryNode_4.right=null;
      for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("right")) {
            f.setAccessible(true); f.set(binaryNode_4, null); break; } }
      // binaryNode_4.left=null;
      for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("left")) {
            f.setAccessible(true); f.set(binaryNode_4, null); break; } }
      java.lang.Object object_2 = (java.lang.Object) clazzjava$lang$Object.newInstance();
      // binaryNode_4.element=object_2;
      for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("element")) {
            f.setAccessible(true); f.set(binaryNode_4, object_2); break; } }
      // binaryNode_1.right=binaryNode_4;
      for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("right")) {
            f.setAccessible(true); f.set(binaryNode_1, binaryNode_4); break; } }
      // binaryNode_1.left=null;
      for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("left")) {
            f.setAccessible(true); f.set(binaryNode_1, null); break; } }
      Object object_1 object_1 = clazzjava$lang$Object.newInstance();
      // binaryNode_1.element=object_1;
      for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
         if (f.getName().equals("element")) {
            f.setAccessible(true); f.set(binaryNode_1, object_1); break; } }
```

Figure 8.12: *KUnit Generated JUnit Test Case for Object Version Insert*

```
// binarySearchTree_0.root=binaryNode_1;
for (Field f : clazzcomparator$BinarySearchTree.getDeclaredFields()) {
  if (f.getName().equals("root")) {
    f.setAccessible(true); f.set(binarySearchTree_0, binaryNode_1); break; } }
// binarySearchTree_0.comparator=comparator_0;
for (Field f : clazzcomparator$BinarySearchTree.getDeclaredFields()) {
  if (f.getName().equals("comparator")) {
    f.setAccessible(true); f.set(binarySearchTree_0, comparator_0); break; } }
this_ = binarySearchTree_0;
java.lang.Object x;
java.lang.Object object_0 = null;
object_0 = (java.lang.Object) clazzjava$lang$Object.newInstance();
x = object_0;
this_.insert(x);
assertEquals(this_, binarySearchTree_0);
// binarySearchTree_0.root==binaryNode_1;
for (Field f : clazzcomparator$BinarySearchTree.getDeclaredFields()) {
  if (f.getName().equals("root")) {
    f.setAccessible(true); assertEquals(binaryNode_1, f.get(binarySearchTree_0)); break;}}
// binaryNode_1.right==binaryNode_4;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("right")) {
    f.setAccessible(true); assertEquals(binaryNode_4, f.get(binaryNode_1)); break; } }
// binaryNode_4.right==null;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("right")) {
    f.setAccessible(true); assertEquals(null, f.get(binaryNode_4)); break; } }
// binaryNode_4.left==null;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("left")) {
    f.setAccessible(true);
    assertEquals(null, f.get(binaryNode_4)); break; } }
// binaryNode_4.element==object_2;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("element")) {
    f.setAccessible(true); assertEquals(object_2, f.get(binaryNode_4)); break; } }
comparator.BinaryNode binaryNode_5 = null;
// binaryNode_5=binaryNode_1.left;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("left")) {
    f.setAccessible(true); binaryNode_5 = (comparator.BinaryNode) f.get(binaryNode_1); break;}}
// binaryNode_5.element==object_0;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("element")) {
    f.setAccessible(true); assertEquals(object_0, f.get(binaryNode_5)); break; } }
// binaryNode_5.left==null;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("left")) {
    f.setAccessible(true); assertEquals(null, f.get(binaryNode_5)); break; } }
// binaryNode_5.right==null;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("right")) {
    f.setAccessible(true); assertEquals(null, f.get(binaryNode_5)); break; } }
// binaryNode_1.element==object_1;
for (Field f : clazzcomparator$BinaryNode.getDeclaredFields()) {
  if (f.getName().equals("element")) {
    f.setAccessible(true); assertEquals(object_1, f.get(binaryNode_1)); break; } }
// binarySearchTree_0.comparator==comparator_0;
for (Field f : clazzcomparator$BinarySearchTree.getDeclaredFields()) {
  if (f.getName().equals("comparator")) {
    f.setAccessible(true); assertEquals(comparator_0, f.get(binarySearchTree_0)); break; } }
assertEquals(x, object_0); } }
```

Figure 8.13: *KUnit Generated JUnit Test Case for Object Version Insert (Continue)*

8.3 Red-black Tree

Red-black tree is a special type of binary search tree with an additional field "color" in each node. The color of each node can be either black or red (hence the name *red-black* tree). The constraints on the colors of nodes make red-black a self balancing tree whose search/insert/remove methods have time complexity of $O(\log n)$. In addition to be a binary search tree, red-black tree has following constraints:

- each node is either red or black;
- the root is black;
- leaves (NULLS) are black;
- both children of each red node are black;
- every path from the root to a leaf contains the same number of black nodes (*black height* is defined as the number of internal black nodes in a path from the root to a leaf).

Figure 8.14 shows an example of red-black tree with black height 2. The red-black implementation

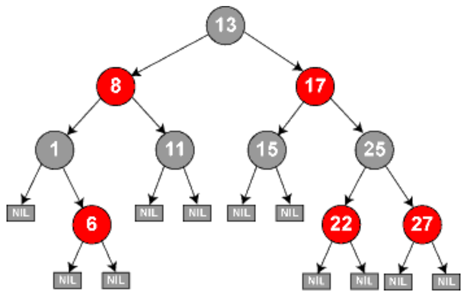


Figure 8.14: *A Red-black Tree Example*

is taken from JDK1.5 library, `java.util.TreeMap`. Figure 8.15 shows the tree node code with invariant method `consistency`. The invariant method `consistency` has following components:

- method `wellConnected` enforces that the parent fields are connected correctly. The source code is shown in Figure 8.16;

- method `redConsistency` enforces that the children of a red node have to be black. The source code is shown in Figure 8.17;
- method `blackConsistency` shown in Figure 8.18 returns true if and only if the current node is black and the black height property holds;
- method `ordered` shown in Figure 8.19 enforces the order property, that is, the keys in the left subtree of a node are less than the key of the node and the keys in the right subtree of a node are greater than the key of the node.

```
    private static final boolean RED = false;

    private static final boolean BLACK = true;

    static public class Entry<V> {
       int key;
       V value;
       Entry<V> left = null;
       Entry<V> right = null;
       Entry<V> parent;
       public boolean consistency() {
          return wellConnected(null) && redConsistency()
                 && blackConsistency() && ordered();
       }
    }
```

Figure 8.15: *Red-black Tree Node*

```
public boolean wellConnected(Entry<V> expectedParent) {
    boolean ok = true;
    if (expectedParent != parent) {
        return false;
    }
    if (right != null) {
        ok = ok && right.wellConnected(this);
    }
    if (left != null) {
        ok = ok && left.wellConnected(this);
    }
    if(right==left && right!=null && left!=null) {
        return false;
    }
    return ok;
}
```

Figure 8.16: *Method wellConnected*

```
/**
 * Returns true if no red node in subtree has red children
 */
protected boolean redConsistency() {
    boolean ret = true;
    if (color == RED && ((left != null && left.color == RED)
                        || (right != null && right.color == RED)))
        return false;
    if (left != null) {
        ret = ret && left.redConsistency();
    }
    if (right != null) {
        ret = ret && right.redConsistency();
    }
    return ret;
}
```

Figure 8.17: *Method redConsistency*

```
protected boolean blackConsistency() {
    if (color != BLACK){ // root must be black
        return false;
    }
    // the number of black nodes on way to any leaf must be same
    if (!consistentlyBlackHeight(blackHeight())) {
        return false;
    }
    return true;
}
/**
 * Returns the black height of this subtree.
 */
protected int blackHeight() {
    int ret = 0;
    if (color == BLACK) {
        ret = 1;
    }
    if (left != null) {
        ret += left.blackHeight();
    }
    return ret;
}
protected boolean consistentlyBlackHeight(int height) {
    boolean ret = true;
    if (color == BLACK)
        height--;
    if (left == null) {
        ret = ret && (height == 0);
    } else {
        ret = ret && (left.consistentlyBlackHeight(height));
    }
    if (right == null) {
        ret = ret && (height == 0);
    } else {
        ret = ret && (right.consistentlyBlackHeight(height));
    }
    return ret;
}
```

Figure 8.18: *Method blackConsistency*

```
boolean ordered() {
    return ordered(this,new Range());
}
boolean ordered(Entry<V> t, Range range) {
    if(t == null) {
        return true;
    }
    if(!range.inRange(t.key)) {
        return false;
    }
    boolean ret=true;
    ret = ret && ordered(t.left,range.setUpper(t.key));
    ret = ret && ordered(t.right,range.setLower(t.key));
    return ret;
}
```

Figure 8.19: *Method ordered*

The `remove` method of TreeMap (integer version) is shown in Figure 8.20. The pre/postconditions essentially enforce the invariant `root.consistency()`. Figure 8.21 shows the corresponding object version of the `remove` method. The specification for the object version includes region specification to help the invariant checking. Similar to binary search tree, the region specification essentially specifies that the tree structure is in region 0 and the keys are in region 1 and values are in region 2. Region 0 can access regions 1 and 2 but regions 1 and 2 can not access region 0.

```
@Assertion(value = { @Case(pre = "(root␣==␣null␣||␣root.consistency())&&
␣␣␣␣␣␣␣␣size==realSize()", post = "(root␣==␣null␣||␣root.consistency())&&
␣␣␣␣␣␣␣␣size==realSize()") })
    public V remove(int key) {
        Entry<V> p = getEntry(key);
        if (p == null)
            return null;

        V oldValue = p.value;
        deleteEntry(p);
        return oldValue;
    }
```

Figure 8.20: *Remove Method of Integer Version TreeMap*

8.3.1 Results

Table 8.3 shows the Kiasan result for the `remove` method of red-black tree. All the machine settings and table arrangements are the same as the insertion sort example. Similar to binary search tree, for each k, we essentially check all the red-black tree with height less than k. Therefore, the

```
    @Assertion(value = { @Case(
        pre = "(Region.setObjReg(this.root,0)␣&&
␣␣␣␣␣␣␣␣␣␣␣␣Region.setFldReg(\"comparator.TreeMap$Entry\",\"left\",0,0)␣&&
␣␣␣␣␣␣␣␣␣␣␣␣Region.setFldReg(\"comparator.TreeMap$Entry\",\"right\",0,0)␣&&
␣␣␣␣␣␣␣␣␣␣␣␣Region.setFldReg(\"comparator.TreeMap$Entry\",\"parent\",0,0)␣&&
␣␣␣␣␣␣␣␣␣␣␣␣Region.setFldReg(\"comparator.TreeMap$Entry\",\"key\",0,1)␣&&
␣␣␣␣␣␣␣␣␣␣␣␣Region.setObjReg(key,1)␣&&␣Region.setFldReg(\"comparator.TreeMap$Entry\",\"value\",0,2)&&
␣␣␣␣␣␣␣␣␣␣␣␣comparator!=null␣&&␣(root␣==␣null␣||␣root.consistency(comparator)))&&
␣␣␣␣␣␣␣␣␣␣␣␣(size==realSize())",
        post = "(root␣==␣null␣||␣root.consistency(comparator))") })
    public V remove(@NonNull Object key) {
        Entry<K, V> p = getEntry(key);
        if (p == null)
            return null;

        V oldValue = p.value;
        deleteEntry(p);
        return oldValue;
    }
```

Figure 8.21: *Remove Method of Object Version TreeMap*

Example	k	Pre	B-Post	A-Post	States	Time
	1	2	5	5	1.4k	0:00.9/0:00.1
`remove()`	2	6	43	43	34.7k	0:07.3/0:04.2
`(int)`	3	31	579	579	1M	5:25.9/4:17.5
	1	2	5	5	1.9k	0:00.9/0:00.1
`remove()`	2	6	43	43	40k	0:08.3/0:04.7
`(Comparator)`	3	31	579	579	1.3M	6:01.5/4:40.5

Table 8.3: *Kiasan Result for Red-black Tree Remove Method*

number of pre-states with k corresponds to the number of non-isomorphic red-black trees with height less than k. The number of post-states equals to $(RBT_n \times \mathbb{Z})/R'$ as shown in Section 4.3. The numbers of pre/post-states are optimal in the sense that they match the minimal numbers of cases of theoretical calculation shown in Section 4.3.

The KUnit results including coverage report for red-black tree are presented in Chapter 9.

Chapter 9

Experiments

This chapter presents a comprehensive experimental study for Kiasan/KUnit and comparisons with similar tools such as jCUTE and JPF. Section 9.1 shows the examples used in the study and Kiasan/KUnit results. Section 9.2 presents the `java.util.TreeMap` coverage report. Comparisons to jCUTE and JPF are presented in Sections 9.3 and 9.4 respectively.

Acknowledgments

This chapter presents a complete experimental study that has been partly published in two papers: first titled titled "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs" by Xianghua Deng, Robby, and John Hatcliff to appear in the Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods [25]; second title " Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems" by Xianghua Deng, Robby, and John Hatcliff to appear in the Proceeding of Testing: Academic & Industrial Conference Practice And Research Techniques (TAIC PART) 2007 [24].

9.1 Examples and Results

To evaluate the effectiveness of Kiasan/KUnit, we have performed a comparative study on twenty three examples. The examples are

- ABS, an absolute value method;
- AvlTree, an AVL tree implementation taken from the data structure textbook [63];
- DoubleLinkedList, a double listed list implementation taken from `java.util.LinkedList`;
- GC, the marking phase of the mark and sweep garbage collection algorithm; It is adapted from a TVLA [42] example.
- TC, a triangle classification;
- StackAr, an array implementation of stack taken from the data structure textbook [63];
- AP, an array partition example which partitions an array into two parts: all elements in one part are less than or equal to the pivot and elements in the other part are greater than the pivot; The example is taken from [6] which is taken from [8].
- BinaryHeap, a binary heap example taken from the data structure textbook [63];
- StackLi, a list implementation of stack taken from the data structure textbook [63];
- Sort, a sorting example that contains insertion sort and shell sort and is taken from the data structure textbook [63];
- BinarySearchTree, a binary search tree example taken from the data structure textbook [63];
- LinkedList, a merge example which takes two sorted linked lists and merges them together;
- TreeMap, a red-black tree implementation taken from JDK 1.5 library, `java.util.TreeMap`;
- DisjSetsFast, a fast disjoint set implementation taken from the data structure textbook [63];
- AP(I), the integer version of array partition;
- BinaryHeap(I), the integer version of the binary heap example;
- LL(I), the integer version of the merge example;

- Sort(I), the integer version of the sorting example.
- AvlTree(I), the integer version of AVL tree example;
- BinarySearchTree(I), the integer version of the binary search tree example;
- TreeMap(I), the integer version of the red-black tree example;
- DisjSets, an original disjoint set implementation taken from the data structure textbook [63];
- Vector, the vector implementation taken from JDK 1.4 library, `java.util.Vector`.

The total example collection contains about 150 (public and helper) methods. For each module considered, the methodology that we followed includes writing a single invariant (`repOK`) that characterizes the correctness of the data structure representation. Using the invariant information, our approach guarantees that all generated tests have test inputs that satisfy the invariant, and the output of each test is automatically checked against the invariant. Additional lightweight contract information for reference non-NULL-ness was also used. Our approach generates tests for the *public* methods of each class. Private helper methods are covered in the path exploration and coverage goals, but only as they are invoked from public methods. Test cases are not generated directly for private helper methods because the class invariant sometimes does not hold at the pre/post-states of private methods (as these are intended to work at intermediate private states in which the data structure invariant does not hold).

Tables 9.1, 9.2, and 9.3 show the the experiment results with lazier# initialization. For comparison purpose, we also list the results for lazier initialization (9.4, 9.5, and 9.6) and lazy initialization (9.7, 9.8, and 9.9).

All the experiments are conducted in a 2.4GHz Opteron Linux workstation with 512MB Java heap. Recall that Kiasan performs a per-method compositional analysis (similar to ESC/Java), and moreover, a bound of $k = 2$ is almost always sufficient for achieving 100% feasible branch coverage. So the results indicate the feasibility of Kiasan in actual development for code similar to these examples.

Class		*k*	Test Cases		Branch	Bytecode	Total	CVC	$\Rightarrow_{\mathcal{B}}^{-1}$ &	JUnit	GraphViz
	Method		Generated	States	Coverage	Coverage	(-dot)	Lite	POOC	Gen.	dot (+)
ABS	abs	1	2	60	2/2=100%	8/8=100%	0.2s	0.0s	0.1s	0.0s	0.4s
AvlTree	find	1	4	1864	10/10=100%	60/60=100%	1.4s	0.0s	0.3s	0.2s	1.2s
		2	21	18800	10/10=100%	60/60=100%	9.3s	2.5s	1.5s	0.7s	3.4s
		3	190	351798	10/10=100%	60/60=100%	3.8m	3.0m	18.6s	1.7s	24.9s
	findMax	1	2	1132	7/8=87%	38/42=90%	0.8s	0.2s	0.1s	0.0s	0.6s
		2	5	7127	8/8=100%	41/42=97%	3.7s	0.8s	0.4s	0.0s	1.1s
		3	20	85445	8/8=100%	41/42=97%	57.3s	46.7s	2.5s	0.6s	3.9s
	findMin	1	2	1132	7/8=87%	38/42=90%	0.8s	0.2s	0.1s	0.0s	0.4s
		2	5	7127	8/8=100%	41/42=97%	3.9s	1.4s	0.3s	0.1s	1.6s
		3	20	85445	8/8=100%	41/42=97%	59.9s	49.4s	2.1s	1.0s	3.4s
	insert	1	4	3053	12/18=66%	119/160=74%	1.3s	0.1s	0.2s	0.1s	0.9s
		2	21	25702	18/18=100%	270/270=100%	8.8s	2.5s	1.5s	0.2s	3.4s
		3	190	422049	18/18=100%	270/270=100%	4.1m	3.2m	24.1s	1.3s	27.1s
DoubleLinkedList	addBefore	1	1	364	1/1=100%	31/31=100%	0.3s	0.0s	0.1s	0.0s	0.2s
		2	1	608	1/1=100%	31/31=100%	0.5s	0.0s	0.1s	0.1s	0.2s
		3	1	914	1/1=100%	31/31=100%	0.6s	0.1s	0.1s	0.0s	0.4s
	clear	1	1	223	1/1=100%	19/19=100%	0.3s	0.0s	0.1s	0.0s	0.2s
		2	2	507	1/1=100%	19/19=100%	0.6s	0.0s	0.1s	0.0s	0.2s
		3	3	832	1/1=100%	19/19=100%	1.0s	0.1s	0.2s	0.1s	0.6s
	indexOf	1	2	373	4/10=40%	28/49=57%	0.3s	0.0s	0.1s	0.0s	0.5s
		2	6	1130	10/10=100%	49/49=100%	1.2s	0.5s	0.1s	0.0s	1.9s
		3	16	3139	10/10=100%	49/49=100%	3.9s	1.3s	0.4s	0.1s	2.5s
	lastIndexOf	1	2	374	4/10=40%	29/50=58%	0.6s	0.0s	0.1s	0.1s	0.4s
		2	6	1132	10/10=100%	50/50=100%	1.2s	0.2s	0.3s	0.0s	1.6s
		3	14	2783	10/10=100%	50/50=100%	2.6s	0.7s	0.6s	0.2s	2.7s
	remove	1	2	371	4/10=40%	26/53=49%	0.7s	0.0s	0.1s	0.0s	0.3s
		2	6	1100	11/12=91%	81/86=94%	1.8s	0.4s	0.4s	0.0s	1.0s
		3	16	3039	11/12=91%	81/86=94%	3.1s	0.2s	1.5s	0.2s	3.0s
	removeLast	1	1	152	1/2=50%	21/46=45%	0.3s	0.1s	0.0s	0.0s	0.0s
		2	2	453	2/2=100%	46/46=100%	0.6s	0.1s	0.1s	0.0s	0.2s
		3	3	839	2/2=100%	46/46=100%	0.8s	0.1s	0.1s	0.0s	0.4s
	toArray	1	1	162	1/2=50%	18/27=66%	0.3s	0.0s	0.1s	0.1s	0.0s
		2	2	399	2/2=100%	27/27=100%	0.5s	0.1s	0.1s	0.0s	0.2s
		3	3	691	2/2=100%	27/27=100%	0.8s	0.2s	0.1s	0.0s	0.5s
GC	Mark	1	306	222149	12/12=100%	64/64=100%	44.8s	2.2s	10.8s	1.3s	59.6s
TC	classify	1	15	404	16/16=100%	54/54=100%	1.7s	0.2s	0.5s	0.5s	2.4s
StackAr	pop	1	2	104	4/4=100%	32/32=100%	0.4s	0.0s	0.1s	0.1s	0.4s
		2	2	104	4/4=100%	32/32=100%	0.3s	0.1s	0.1s	0.1s	0.3s
		3	2	104	4/4=100%	32/32=100%	0.3s	0.0s	0.1s	0.1s	0.3s
	push	1	2	120	4/4=100%	36/36=100%	0.4s	0.1s	0.1s	0.1s	0.2s
		2	2	120	4/4=100%	36/36=100%	0.3s	0.0s	0.1s	0.0s	0.3s
		3	2	120	4/4=100%	36/36=100%	0.3s	0.0s	0.1s	0.1s	0.2s
AP	partition	1	1	150	1/10=10%	18/68=26%	0.3s	0.0s	0.1s	0.0s	0.3s
		2	3	389	9/10=90%	50/68=73%	0.8s	0.3s	0.1s	0.1s	0.5s
		3	7	1006	10/10=100%	68/68=100%	1.9s	0.6s	0.5s	0.2s	1.8s
BinaryHeap	deleteMin	1	2	288	6/14=42%	71/120=59%	0.7s	0.1s	0.3s	0.0s	0.4s
		2	3	488	6/14=42%	71/120=59%	1.0s	0.1s	0.1s	0.0s	0.5s
		3	5	890	10/14=71%	101/120=84%	1.9s	0.7s	0.6s	0.1s	0.8s
		4	8	1585	13/14=92%	117/120=97%	3.2s	1.7s	0.5s	0.0s	1.7s
	insert	1	3	336	8/8=100%	65/69=94%	0.9s	0.1s	0.1s	0.1s	0.2s
		2	6	650	8/8=100%	69/69=100%	1.5s	0.4s	0.2s	0.0s	1.1s
		3	9	1098	8/8=100%	69/69=100%	2.3s	1.0s	0.4s	0.3s	1.7s
		4	13	1853	8/8=100%	69/69=100%	4.4s	1.9s	0.4s	0.4s	1.8s
Sort	insertionSort	1	2	178	1/6=16%	9/44=20%	0.5s	0.0s	0.1s	0.2s	0.4s
		2	4	376	6/6=100%	44/44=100%	0.8s	0.3s	0.1s	0.0s	1.2s
		3	10	1088	6/6=100%	44/44=100%	2.7s	1.7s	0.3s	0.0s	1.4s
	shellsort	1	2	180	1/8=12%	10/61=16%	0.4s	0.0s	0.1s	0.0s	0.3s
		2	4	406	8/8=100%	61/61=100%	1.0s	0.2s	0.1s	0.1s	0.6s
		3	10	1192	8/8=100%	61/61=100%	3.2s	1.4s	0.4s	0.0s	1.4s

Table 9.1: *Lazier# Experiment Data (1); s – seconds; m – minutes*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_{\mathcal{B}}^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
BinarySearchTree	find	1	4	1162	8/8=100%	65/65=100%	1.2s	0.0s	0.2s	0.4s	0.6s
		2	21	10443	8/8=100%	65/65=100%	6.5s	1.0s	1.1s	0.6s	3.1s
		3	236	212296	8/8=100%	65/65=100%	1.7m	1.1m	12.0s	1.5s	33.9s
	findMax	1	2	689	5/6=83%	29/32=90%	0.7s	0.0s	0.2s	0.0s	0.5s
		2	5	3719	6/6=100%	32/32=100%	2.3s	0.4s	0.2s	0.1s	0.9s
		3	26	43215	6/6=100%	32/32=100%	27.0s	20.2s	1.3s	0.4s	4.0s
	findMin	1	2	688	5/6=83%	30/37=81%	0.6s	0.0s	0.1s	0.1s	0.5s
		2	5	3729	6/6=100%	37/37=100%	2.2s	0.3s	0.2s	0.2s	1.4s
		3	26	43400	6/6=100%	37/37=100%	27.0s	19.4s	1.3s	0.6s	4.4s
	insert	1	4	1621	6/6=100%	63/63=100%	1.7s	0.1s	0.3s	0.1s	1.0s
		2	21	12551	6/6=100%	63/63=100%	6.2s	1.7s	1.0s	0.1s	4.9s
		3	236	234595	6/6=100%	63/63=100%	1.8m	1.2m	13.4s	1.3s	31.6s
	remove	1	4	1001	8/12=66%	68/95=71%	1.0s	0.0s	0.1s	0.2s	1.3s
		2	21	9254	14/16=87%	104/113=92%	7.3s	3.4s	1.2s	0.1s	3.1s
		3	236	197738	15/16=93%	111/113=98%	1.6m	1.0m	12.0s	1.4s	35.4s
TreeMap	get	1	4	1199	9/10=90%	60/67=89%	1.2s	0.6s	0.2s	0.0s	0.7s
		2	28	17440	9/10=90%	60/67=89%	8.4s	3.3s	0.8s	0.2s	6.4s
		3	331	470913	9/10=90%	60/67=89%	2.3m	1.5m	13.9s	1.3s	50.8s
	lastKey	1	2	657	5/6=83%	32/35=91%	0.5s	0.1s	0.2s	0.0s	0.2s
		2	6	7614	6/6=100%	35/35=100%	2.8s	0.5s	0.3s	0.1s	1.3s
		3	31	204738	6/6=100%	35/35=100%	27.8s	16.5s	1.1s	0.3s	5.2s
	put	1	4	1871	12/32=37%	133/371=35%	1.2s	0.2s	0.2s	0.0s	0.7s
		2	28	22872	40/52=76%	470/500=94%	8.6s	2.7s	1.1s	0.7s	4.9s
		3	331	530005	42/52=80%	478/500=95%	2.2m	1.3m	15.9s	1.8s	58.0s
	remove	1	4	1110	13/34=38%	105/216=48%	1.0s	0.2s	0.2s	0.1s	0.8s
		2	28	17081	45/74=60%	347/586=59%	7.2s	2.6s	1.1s	0.3s	6.2s
		3	331	472985	70/86=81%	640/684=93%	2.2m	1.4m	14.8s	1.5s	52.6s
DisjSetsFast	Find	1	1	281	1/2=50%	14/28=50%	0.4s	0.1s	0.1s	0.0s	0.3s
		2	7	1268	2/2=100%	28/28=100%	2.4s	1.1s	0.4s	0.1s	1.3s
		3	55	6485	2/2=100%	28/28=100%	13.1s	6.7s	2.3s	0.2s	8.0s
	union	2	6	1643	6/6=100%	62/62=100%	2.5s	1.2s	0.4s	0.0s	1.1s
		3	60	10096	6/6=100%	62/62=100%	17.7s	12.2s	3.0s	0.1s	9.4s
AP(I)	partition	1	1	103	3/10=30%	27/62=43%	0.4s	0.1s	0.1s	0.0s	0.1s
		2	3	252	9/10=90%	44/62=70%	0.6s	0.2s	0.1s	0.0s	0.5s
		3	7	691	10/10=100%	62/62=100%	1.5s	0.5s	0.2s	0.0s	0.7s
BinaryHeap(I)	deleteMin	1	2	246	6/14=42%	71/114=62%	0.6s	0.2s	0.1s	0.1s	0.3s
		2	3	406	6/14=42%	71/114=62%	0.8s	0.3s	0.1s	0.0s	0.5s
		3	5	727	10/14=71%	98/114=85%	1.3s	0.5s	0.1s	0.1s	1.0s
		4	8	1287	13/14=92%	111/114=97%	1.6s	0.5s	0.2s	0.1s	1.8s
	findMin	1	2	190	4/4=100%	21/21=100%	0.3s	0.1s	0.1s	0.0s	0.4s
		2	3	328	4/4=100%	21/21=100%	0.6s	0.1s	0.1s	0.0s	0.4s
		3	4	513	4/4=100%	21/21=100%	1.0s	0.3s	0.1s	0.1s	0.9s
	insert	1	2	205	6/6=100%	49/53=92%	0.7s	0.1s	0.1s	0.1s	0.2s
		2	5	479	6/6=100%	53/53=100%	1.1s	0.3s	0.1s	0.2s	0.6s
		3	8	836	6/6=100%	53/53=100%	1.7s	0.4s	0.2s	0.2s	0.8s
		4	12	1446	6/6=100%	53/53=100%	3.1s	0.9s	0.4s	0.5s	1.9s
LL(I)	merge	1	1	424	3/10=30%	29/72=40%	0.4s	0.0s	0.1s	0.0s	0.2s
		2	5	1973	10/10=100%	72/72=100%	1.6s	0.2s	0.2s	0.2s	1.0s
		3	19	9284	10/10=100%	72/72=100%	3.6s	1.0s	0.5s	0.1s	5.0s
Sort(I)	insertionSort	1	1	88	3/6=50%	25/41=60%	0.3s	0.0s	0.1s	0.1s	0.2s
		2	3	218	6/6=100%	41/41=100%	0.5s	0.2s	0.1s	0.0s	0.5s
		3	9	770	6/6=100%	41/41=100%	1.7s	0.6s	0.2s	0.0s	1.8s
	shellsort	1	1	98	4/8=50%	35/58=60%	0.3s	0.1s	0.1s	0.0s	0.1s
		2	3	246	8/8=100%	58/58=100%	0.9s	0.4s	0.1s	0.0s	0.2s
		3	9	828	8/8=100%	58/58=100%	2.7s	1.4s	0.3s	0.1s	1.0s
StackLi	pop	1	2	189	4/4=100%	25/25=100%	0.3s	0.0s	0.0s	0.1s	0.1s
		2	3	377	4/4=100%	25/25=100%	0.4s	0.0s	0.1s	0.0s	0.4s
		3	4	662	4/4=100%	25/25=100%	0.7s	0.1s	0.1s	0.1s	0.6s
	push	1	2	374	1/1=100%	10/10=100%	0.5s	0.0s	0.1s	0.1s	0.3s
		2	3	687	1/1=100%	10/10=100%	1.0s	0.0s	0.1s	0.1s	0.4s
		3	4	1119	1/1=100%	10/10=100%	1.1s	0.0s	0.3s	0.0s	0.7s

Table 9.2: *Lazier# Experiment Data (2); s – seconds; m – minutes*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_{\mathcal{B}}^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
AvlTree(I)	find	1	4	1438	10/10=100%	54/54=100%	1.4s	0.1s	0.4s	0.0s	0.9s
		2	21	14101	10/10=100%	54/54=100%	5.5s	1.0s	0.9s	0.2s	4.1s
		3	190	256694	10/10=100%	54/54=100%	50.0s	17.2s	10.6s	0.9s	33.3s
	findMax	1	2	933	4/4=100%	27/27=100%	0.7s	0.0s	0.2s	0.0s	0.4s
		2	5	5890	4/4=100%	27/27=100%	2.1s	0.4s	0.3s	0.0s	2.1s
		3	20	72006	4/4=100%	27/27=100%	12.8s	5.7s	1.1s	0.5s	4.2s
	insert	1	4	2343	12/18=66%	113/148=76%	1.5s	0.0s	0.4s	0.0s	0.7s
		2	21	19682	18/18=100%	234/234=100%	6.2s	2.1s	0.8s	0.1s	3.7s
		3	190	315704	18/18=100%	234/234=100%	56.2s	20.5s	13.5s	0.7s	32.5s
BinarySearchTree(I)	find	1	4	1044	1/1=100%	12/12=100%	0.8s	0.0s	0.1s	0.0s	1.1s
		2	21	10016	1/1=100%	12/12=100%	4.1s	0.7s	0.7s	0.2s	4.6s
		3	236	207937	1/1=100%	12/12=100%	48.1s	14.8s	7.6s	0.9s	43.7s
	findMax	1	2	601	3/4=75%	21/24=87%	0.7s	0.0s	0.1s	0.2s	0.3s
		2	5	3432	4/4=100%	24/24=100%	2.0s	0.1s	0.3s	0.1s	1.1s
		3	26	39950	4/4=100%	24/24=100%	9.8s	4.0s	0.8s	0.3s	4.7s
	findMin	1	2	600	3/4=75%	22/29=75%	0.4s	0.1s	0.1s	0.0s	0.4s
		2	5	3442	4/4=100%	29/29=100%	2.6s	0.1s	0.2s	0.1s	0.8s
		3	26	40135	4/4=100%	29/29=100%	10.1s	3.6s	1.0s	0.2s	4.8s
	insert	1	4	1516	6/6=100%	53/53=100%	1.5s	0.0s	0.3s	0.0s	0.5s
		2	21	12173	6/6=100%	53/53=100%	5.3s	0.9s	0.8s	0.1s	4.2s
		3	236	230681	6/6=100%	53/53=100%	50.5s	17.4s	11.2s	1.5s	43.1s
	remove	1	4	884	8/12=66%	58/85=68%	0.9s	0.1s	0.1s	0.1s	0.9s
		2	21	8824	14/16=87%	94/103=91%	5.2s	0.6s	0.9s	0.2s	3.7s
		3	236	193182	15/16=93%	101/103=98%	46.7s	15.8s	8.8s	0.6s	43.2s
TreeMap(I)	get	1	4	740	8/8=100%	44/44=100%	0.8s	0.1s	0.1s	0.0s	0.6s
		2	28	8205	8/8=100%	44/44=100%	6.6s	2.1s	1.9s	0.1s	6.1s
		3	331	212319	8/8=100%	44/44=100%	1.3m	28.1s	25.4s	1.1s	1.1m
	lastKey	1	2	610	5/6=83%	32/35=91%	0.6s	0.0s	0.1s	0.1s	0.8s
		2	6	7211	6/6=100%	35/35=100%	3.5s	0.5s	0.3s	0.0s	1.0s
		3	31	201663	6/6=100%	35/35=100%	29.3s	17.6s	1.1s	0.6s	4.2s
	put	1	4	2781	15/34=44%	135/366=36%	1.5s	0.1s	0.2s	0.0s	1.0s
		2	28	35344	43/54=79%	472/495=95%	9.2s	2.4s	1.5s	0.3s	5.3s
		3	331	813226	45/54=83%	480/495=96%	2.5m	1.2m	19.9s	2.0s	47.7s
	remove	1	4	1542	12/32=37%	89/193=46%	1.3s	0.4s	0.2s	0.1s	0.9s
		2	28	27324	44/72=61%	331/563=58%	8.4s	3.6s	1.2s	0.2s	4.9s
		3	331	752174	69/84=82%	624/661=94%	2.3m	1.2m	19.8s	1.4s	48.5s
DisjSets	Find	1	1	266	1/2=50%	14/23=60%	0.5s	0.3s	0.1s	0.0s	0.0s
		2	7	1223	2/2=100%	23/23=100%	2.1s	0.7s	0.6s	0.0s	1.2s
		3	55	6301	2/2=100%	23/23=100%	11.8s	7.2s	2.1s	0.2s	7.3s
	union	2	2	1294	2/2=100%	23/23=100%	1.6s	0.6s	0.1s	0.0s	0.4s
		3	20	6938	2/2=100%	23/23=100%	8.3s	4.6s	1.2s	0.1s	2.4s
LL	merge	1	1	732	3/10=30%	29/76=38%	1.6s	0.1s	1.2s	0.0s	0.3s
		2	5	3631	10/10=100%	76/76=100%	2.2s	0.0s	0.4s	0.1s	2.2s
		3	19	17754	10/10=100%	76/76=100%	7.4s	1.1s	1.2s	0.6s	3.4s
Vector	add	1	3	354	8/8=100%	77/77=100%	0.7s	0.3s	0.1s	0.0s	0.1s
		2	5	472	8/8=100%	77/77=100%	1.5s	0.6s	0.1s	0.3s	0.6s
		3	7	590	8/8=100%	77/77=100%	1.5s	0.7s	0.2s	0.1s	1.1s
	ensureCapacity	1	9	610	8/8=100%	62/62=100%	1.4s	0.2s	0.2s	0.4s	1.9s
		2	13	826	8/8=100%	62/62=100%	2.0s	0.7s	0.2s	0.1s	2.6s
		3	17	1042	8/8=100%	62/62=100%	3.5s	2.1s	0.3s	0.3s	2.4s
	indexOf	1	6	438	10/10=100%	41/41=100%	1.2s	0.5s	0.1s	0.0s	1.2s
		2	7	486	10/10=100%	41/41=100%	1.3s	0.7s	0.2s	0.0s	1.2s
		3	7	486	10/10=100%	41/41=100%	0.8s	0.2s	0.2s	0.1s	1.6s
	insertElementAt	1	3	469	11/12=91%	105/112=93%	1.2s	0.3s	0.1s	0.1s	0.3s
		2	8	836	11/12=91%	106/112=94%	1.2s	0.5s	0.2s	0.1s	1.8s
		3	15	1389	11/12=91%	106/112=94%	4.1s	1.8s	0.7s	0.1s	2.0s
	lastIndexOf	1	7	445	12/12=100%	48/48=100%	1.0s	0.5s	0.1s	0.1s	1.0s
		2	17	975	12/12=100%	48/48=100%	2.0s	0.6s	0.4s	0.4s	3.0s
		3	39	2141	12/12=100%	48/48=100%	6.2s	1.7s	2.7s	0.2s	5.4s
	removeElementAt	1	3	197	7/8=87%	70/71=98%	0.7s	0.3s	0.1s	0.1s	0.3s
		2	4	257	8/8=100%	71/71=100%	0.8s	0.2s	0.1s	0.0s	0.4s
		3	5	318	8/8=100%	71/71=100%	1.0s	0.2s	0.2s	0.0s	1.0s

Table 9.3: *Lazier# Experiment Data (3); s – seconds; m – minutes*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_{\mathcal{A}}^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
ABS	abs	1	2	60	2/2=100%	8/8=100%	0.2s	0.0s	0.1s	0.1s	0.3s
AvlTree	find	1	4	2420	10/10=100%	60/60=100%	1.3s	0.1s	0.3s	0.1s	1.4s
		2	21	23807	10/10=100%	60/60=100%	7.8s	2.3s	1.6s	0.3s	3.9s
		3	190	459718	10/10=100%	60/60=100%	2.8m	2.0m	22.6s	1.1s	23.5s
	findMax	1	2	1457	7/8=87%	38/42=90%	0.9s	0.0s	0.2s	0.2s	0.8s
		2	5	8733	8/8=100%	41/42=97%	2.5s	0.4s	0.4s	0.0s	1.6s
		3	20	107574	8/8=100%	41/42=97%	20.7s	10.6s	2.2s	0.4s	2.5s
	findMin	1	2	1457	7/8=87%	38/42=90%	1.0s	0.0s	0.2s	0.1s	0.8s
		2	5	8733	8/8=100%	41/42=97%	2.9s	0.3s	0.3s	0.1s	1.1s
		3	20	107574	8/8=100%	41/42=97%	20.4s	10.3s	2.1s	0.4s	2.7s
	insert	1	4	3841	12/18=66%	119/160=74%	2.0s	0.1s	0.5s	0.1s	1.4s
		2	21	31905	18/18=100%	270/270=100%	7.3s	2.0s	1.4s	0.4s	4.4s
		3	190	542929	18/18=100%	270/270=100%	3.2m	2.2m	26.4s	1.3s	22.8s
DoubleLinkedList	addBefore	1	2	1010	1/1=100%	31/31=100%	0.7s	0.0s	0.1s	0.0s	0.5s
		2	6	2922	1/1=100%	31/31=100%	1.8s	0.2s	0.2s	0.1s	1.3s
		3	12	5846	1/1=100%	31/31=100%	2.6s	0.8s	0.3s	0.1s	4.2s
	clear	1	1	226	1/1=100%	19/19=100%	0.2s	0.0s	0.1s	0.0s	0.0s
		2	2	513	1/1=100%	19/19=100%	0.6s	0.2s	0.1s	0.0s	0.4s
		3	3	841	1/1=100%	19/19=100%	0.8s	0.0s	0.1s	0.1s	0.5s
	indexOf	1	2	668	4/10=40%	28/49=57%	0.7s	0.3s	0.1s	0.0s	0.2s
		2	6	1650	10/10=100%	49/49=100%	1.6s	0.3s	0.2s	0.0s	1.5s
		3	16	3937	10/10=100%	49/49=100%	2.8s	0.2s	0.4s	0.2s	3.7s
	lastIndexOf	1	2	670	4/10=40%	29/50=58%	0.4s	0.0s	0.1s	0.0s	0.3s
		2	6	1654	10/10=100%	50/50=100%	2.0s	0.1s	0.3s	0.3s	0.9s
		3	14	3583	10/10=100%	50/50=100%	4.1s	1.3s	0.3s	0.2s	2.4s
	remove	1	2	664	4/10=40%	26/53=49%	0.9s	0.1s	0.1s	0.1s	0.5s
		2	6	1616	11/12=91%	81/86=94%	2.1s	0.1s	0.2s	0.3s	1.2s
		3	16	3831	11/12=91%	81/86=94%	4.0s	0.2s	0.6s	0.6s	3.2s
	removeLast	1	2	180	1/2=50%	21/46=45%	0.3s	0.0s	0.1s	0.0s	0.2s
		2	4	597	2/2=100%	46/46=100%	1.0s	0.1s	0.2s	0.1s	0.4s
		3	6	1143	2/2=100%	46/46=100%	1.1s	0.1s	0.1s	0.1s	1.0s
	toArray	1	1	165	1/2=50%	18/27=66%	0.2s	0.0s	0.1s	0.0s	0.2s
		2	3	437	2/2=100%	27/27=100%	0.6s	0.0s	0.1s	0.1s	0.8s
		3	7	842	2/2=100%	27/27=100%	1.0s	0.1s	0.2s	0.2s	1.5s
GC	Mark	1	306	222575	12/12=100%	64/64=100%	40.6s	2.0s	8.5s	1.4s	1.2m
TC	classify	1	15	404	16/16=100%	54/54=100%	1.6s	0.3s	0.5s	0.3s	2.7s
StackAr	pop	1	2	105	4/4=100%	32/32=100%	0.4s	0.1s	0.1s	0.1s	0.1s
		2	2	105	4/4=100%	32/32=100%	0.3s	0.1s	0.1s	0.0s	0.3s
		3	2	105	4/4=100%	32/32=100%	0.4s	0.0s	0.1s	0.1s	0.2s
	push	1	4	238	4/4=100%	36/36=100%	0.3s	0.1s	0.1s	0.0s	0.8s
		2	4	238	4/4=100%	36/36=100%	0.3s	0.1s	0.1s	0.0s	0.4s
		3	4	238	4/4=100%	36/36=100%	1.0s	0.4s	0.1s	0.1s	0.3s
AP	partition	1	1	152	1/10=10%	18/68=26%	0.5s	0.0s	0.1s	0.2s	0.0s
		2	3	392	9/10=90%	50/68=73%	0.9s	0.2s	0.3s	0.0s	0.6s
		3	7	1010	10/10=100%	68/68=100%	2.3s	0.8s	0.3s	0.0s	1.1s
BinaryHeap	deleteMin	1	2	290	6/14=42%	71/120=59%	0.7s	0.1s	0.1s	0.2s	0.2s
		2	3	491	6/14=42%	71/120=59%	0.9s	0.4s	0.1s	0.1s	0.5s
		3	5	894	10/14=71%	101/120=84%	1.7s	0.6s	0.1s	0.0s	0.8s
		4	8	1590	13/14=92%	117/120=97%	3.3s	1.7s	0.3s	0.2s	1.2s
	insert	1	3	349	8/8=100%	65/69=94%	0.7s	0.1s	0.1s	0.0s	0.2s
		2	6	664	8/8=100%	69/69=100%	1.8s	0.4s	0.2s	0.1s	0.7s
		3	9	1113	8/8=100%	69/69=100%	2.7s	0.9s	0.4s	0.2s	1.3s
		4	13	1869	8/8=100%	69/69=100%	3.8s	1.8s	0.5s	0.2s	2.0s
Sort	insertionSort	1	2	190	1/6=16%	9/44=20%	0.4s	0.0s	0.1s	0.0s	0.4s
		2	4	389	6/6=100%	44/44=100%	1.0s	0.1s	0.2s	0.0s	0.6s
		3	10	1102	6/6=100%	44/44=100%	2.8s	1.2s	0.3s	0.3s	1.9s
	shellsort	1	2	192	1/8=12%	10/61=16%	0.3s	0.1s	0.1s	0.0s	0.4s
		2	4	419	8/8=100%	61/61=100%	1.1s	0.3s	0.1s	0.0s	0.7s
		3	10	1206	8/8=100%	61/61=100%	3.1s	1.1s	0.5s	0.1s	1.9s

Table 9.4: *Lazier Experiment Data (1); s – seconds; m – minutes*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_{\mathcal{A}}^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
BinarySearchTree	find	1	12	4301	8/8=100%	65/65=100%	3.2s	0.9s	0.5s	0.2s	3.1s
		2	98	57292	8/8=100%	65/65=100%	19.6s	7.4s	3.8s	0.6s	18.9s
		3	1788	1808683	8/8=100%	65/65=100%	25.3m	12.8m	9.4m	8.8s	3.7m
	findMax	1	3	1430	5/6=83%	29/32=90%	0.9s	0.0s	0.1s	0.0s	0.7s
		2	13	12821	6/6=100%	32/32=100%	6.1s	0.9s	0.5s	0.2s	3.1s
		3	131	258323	6/6=100%	32/32=100%	2.8m	2.4m	6.7s	1.0s	17.6s
	findMin	1	3	1428	5/6=83%	30/37=81%	1.2s	0.0s	0.2s	0.1s	0.6s
		2	13	12858	6/6=100%	37/37=100%	5.0s	1.0s	0.8s	0.4s	2.4s
		3	131	259404	6/6=100%	37/37=100%	2.8m	2.4m	7.6s	1.0s	18.2s
	insert	1	12	5521	6/6=100%	63/63=100%	3.0s	0.2s	0.5s	0.1s	2.5s
		2	94	63931	6/6=100%	63/63=100%	19.1s	8.2s	3.2s	0.9s	18.7s
		3	1668	1855571	6/6=100%	63/63=100%	27.3m	13.2m	10.6m	7.5s	3.5m
	remove	1	12	3693	8/12=66%	68/95=71%	2.5s	0.1s	0.4s	0.2s	2.7s
		2	94	49422	14/16=87%	104/113=92%	18.6s	7.7s	2.8s	1.1s	17.4s
		3	1668	1599087	15/16=93%	111/113=98%	20.7m	11.2m	7.0m	6.6s	3.7m
LL	merge	1	1	1007	3/10=30%	29/76=38%	0.6s	0.0s	0.1s	0.0s	0.2s
		2	6	3931	10/10=100%	76/76=100%	2.4s	0.1s	0.3s	0.1s	1.6s
		3	30	18122	10/10=100%	76/76=100%	9.1s	2.5s	2.2s	0.6s	6.5s
TreeMap	get	1	6	2009	9/10=90%	60/67=89%	1.8s	0.3s	0.2s	0.0s	1.2s
		2	40	27489	9/10=90%	60/67=89%	9.6s	3.4s	1.4s	0.5s	11.4s
		3	482	774545	9/10=90%	60/67=89%	3.5m	2.3m	20.5s	2.7s	1.4m
	lastKey	1	2	664	5/6=83%	32/35=91%	0.6s	0.1s	0.1s	0.1s	0.3s
		2	6	7658	6/6=100%	35/35=100%	3.1s	0.5s	0.6s	0.3s	1.6s
		3	31	205430	6/6=100%	35/35=100%	29.4s	17.0s	1.0s	0.3s	4.4s
	put	1	10	4125	12/32=37%	133/371=35%	2.4s	0.1s	0.6s	0.2s	1.5s
		2	78	54221	40/52=76%	470/500=94%	17.0s	9.2s	2.3s	0.5s	16.9s
		3	1978	2676863	42/52=80%	478/500=95%	13.7m	7.8m	2.6m	7.2s	6.0m
	remove	1	5	1721	13/34=38%	105/216=48%	1.9s	0.1s	0.3s	0.5s	1.0s
		2	43	37832	45/74=60%	347/586=59%	12.7s	5.7s	2.3s	0.4s	8.2s
		3	579	1166311	70/86=81%	640/684=93%	8.8m	6.2m	1.4m	2.8s	1.2m
DisjSetsFast	Find	1	1	282	1/2=50%	14/28=50%	0.4s	0.1s	0.1s	0.0s	0.3s
		2	7	1269	2/2=100%	28/28=100%	1.9s	1.1s	0.2s	0.0s	1.4s
		3	55	6486	2/2=100%	28/28=100%	11.2s	5.8s	2.6s	0.2s	8.8s
	union	2	6	1644	6/6=100%	62/62=100%	2.9s	1.1s	0.3s	0.0s	1.0s
		3	60	10097	6/6=100%	62/62=100%	17.9s	12.4s	2.5s	0.2s	9.9s
AP(I)	partition	1	1	112	3/10=30%	35/91=38%	0.4s	0.1s	0.1s	0.0s	0.3s
		2	3	289	9/10=90%	67/91=73%	0.8s	0.1s	0.2s	0.0s	0.5s
		3	7	811	10/10=100%	91/91=100%	1.2s	0.2s	0.1s	0.0s	1.6s
BinaryHeap(I)	deleteMin	1	2	247	6/14=42%	71/114=62%	0.5s	0.1s	0.1s	0.0s	0.3s
		2	3	407	6/14=42%	71/114=62%	0.7s	0.1s	0.1s	0.0s	0.5s
		3	5	728	10/14=71%	98/114=85%	0.9s	0.2s	0.1s	0.1s	1.0s
		4	8	1288	13/14=92%	111/114=97%	1.8s	0.6s	0.2s	0.1s	1.4s
	insert	1	2	206	6/6=100%	49/53=92%	0.4s	0.1s	0.1s	0.0s	0.5s
		2	5	480	6/6=100%	53/53=100%	0.9s	0.3s	0.2s	0.1s	1.0s
		3	8	837	6/6=100%	53/53=100%	1.4s	0.4s	0.2s	0.1s	1.8s
		4	12	1447	6/6=100%	53/53=100%	2.8s	0.8s	0.4s	0.2s	1.6s
LL(I)	merge	1	1	498	3/10=30%	29/72=40%	0.4s	0.0s	0.1s	0.1s	0.2s
		2	6	2072	10/10=100%	72/72=100%	1.7s	0.4s	0.2s	0.0s	1.9s
		3	30	9497	10/10=100%	72/72=100%	7.5s	3.8s	1.0s	0.2s	4.8s
Sort(I)	insertionSort	1	1	88	3/6=50%	25/41=60%	0.4s	0.2s	0.1s	0.0s	0.0s
		2	3	218	6/6=100%	41/41=100%	0.6s	0.1s	0.1s	0.0s	0.5s
		3	9	770	6/6=100%	41/41=100%	1.9s	1.0s	0.4s	0.0s	1.2s
	shellsort	1	1	98	4/8=50%	35/58=60%	0.3s	0.1s	0.1s	0.0s	0.1s
		2	3	246	8/8=100%	58/58=100%	0.8s	0.5s	0.1s	0.0s	0.4s
		3	9	828	8/8=100%	58/58=100%	2.3s	1.3s	0.3s	0.0s	1.2s
DisjSets	Find	1	1	267	1/2=50%	14/23=60%	0.4s	0.1s	0.1s	0.0s	0.6s
		2	7	1224	2/2=100%	23/23=100%	1.8s	0.8s	0.2s	0.0s	1.2s
		3	55	6302	2/2=100%	23/23=100%	10.1s	4.6s	2.1s	0.2s	9.4s
	union	2	2	1295	2/2=100%	23/23=100%	1.5s	0.8s	0.1s	0.0s	0.5s
		3	20	6939	2/2=100%	23/23=100%	7.8s	4.7s	0.9s	0.0s	2.8s

Table 9.5: *Lazier Experiment Data (2); s – seconds; m – minutes*

Class		k	Test Cases		Branch	Bytecode	Total	CVC	$\Rightarrow_{\mathcal{A}}^{-1}$ &	JUnit	GraphViz
	Method		Generated	States	Coverage	Coverage	(-dot)	Lite	POOC	Gen.	dot (+)
AvlTree(I)	find	1	4	1555	10/10=100%	54/54=100%	1.2s	0.2s	0.2s	0.0s	1.2s
		2	21	14534	10/10=100%	54/54=100%	6.4s	1.4s	1.1s	0.2s	4.2s
		3	190	259485	10/10=100%	54/54=100%	52.4s	20.9s	8.7s	0.7s	30.1s
	findMax	1	2	1050	4/4=100%	27/27=100%	0.6s	0.0s	0.1s	0.1s	0.6s
		2	5	6323	4/4=100%	27/27=100%	2.1s	0.2s	0.4s	0.2s	1.7s
		3	20	74797	4/4=100%	27/27=100%	12.6s	5.4s	1.2s	0.1s	3.3s
	findMin	1	2	1050	7/8=87%	38/42=90%	0.6s	0.0s	0.1s	0.1s	0.5s
		2	5	6323	8/8=100%	41/42=97%	2.4s	0.2s	0.2s	0.0s	1.3s
		3	20	74797	8/8=100%	41/42=97%	13.8s	6.9s	1.2s	0.2s	2.5s
	insert	1	4	2460	12/18=66%	113/148=76%	1.3s	0.0s	0.4s	0.2s	0.6s
		2	21	20115	18/18=100%	234/234=100%	5.6s	0.8s	1.2s	0.3s	4.9s
		3	190	318495	18/18=100%	234/234=100%	56.6s	20.5s	11.0s	0.7s	31.2s
BinarySearchTree(I)	find	1	4	1165	1/1=100%	12/12=100%	1.2s	0.0s	0.3s	0.0s	0.5s
		2	21	10453	1/1=100%	12/12=100%	4.5s	0.9s	0.7s	0.1s	3.6s
		3	236	210732	1/1=100%	12/12=100%	51.3s	18.7s	10.0s	1.8s	41.3s
	findMax	1	2	722	3/4=75%	21/24=87%	0.6s	0.0s	0.1s	0.0s	0.4s
		2	5	3869	4/4=100%	24/24=100%	1.8s	0.2s	0.2s	0.1s	1.4s
		3	26	42745	4/4=100%	24/24=100%	10.3s	5.2s	0.7s	0.1s	4.3s
	findMin	1	2	721	3/4=75%	22/29=75%	0.5s	0.0s	0.1s	0.1s	0.3s
		2	5	3879	4/4=100%	29/29=100%	1.7s	0.1s	0.2s	0.1s	1.0s
		3	26	42930	4/4=100%	29/29=100%	9.7s	4.0s	0.8s	0.1s	4.2s
	insert	1	4	1637	6/6=100%	53/53=100%	1.5s	0.0s	0.5s	0.1s	0.8s
		2	21	12610	6/6=100%	53/53=100%	5.5s	0.8s	0.6s	0.4s	4.0s
		3	236	233476	6/6=100%	53/53=100%	53.1s	21.4s	7.9s	1.0s	41.5s
	remove	1	4	1005	8/12=66%	58/85=68%	1.1s	0.0s	0.3s	0.1s	1.0s
		2	21	9261	14/16=87%	94/103=91%	4.8s	0.7s	1.1s	0.2s	3.5s
		3	236	195977	15/16=93%	101/103=98%	48.5s	17.4s	9.5s	1.6s	41.3s
TreeMap(I)	get	1	5	698	8/8=100%	44/44=100%	0.9s	0.0s	0.4s	0.0s	1.3s
		2	39	8083	8/8=100%	44/44=100%	6.3s	1.3s	2.9s	0.1s	7.0s
		3	739	286729	8/8=100%	44/44=100%	2.3m	45.2s	1.0m	3.0s	3.0m
	lastKey	1	2	615	5/6=83%	32/35=91%	0.8s	0.5s	0.1s	0.0s	0.4s
		2	6	7243	6/6=100%	35/35=100%	3.1s	0.5s	0.3s	0.2s	1.2s
		3	81	522702	6/6=100%	35/35=100%	1.4m	56.0s	3.4s	0.5s	9.0s
	put	1	10	6137	15/34=44%	135/366=36%	3.0s	0.1s	0.6s	0.0s	2.0s
		2	78	86567	43/54=79%	472/495=95%	19.3s	6.3s	2.8s	0.4s	15.0s
		3	1481	3308886	45/54=83%	480/495=96%	11.2m	5.9m	1.8m	4.2s	3.9m
	remove	1	5	1623	12/32=37%	89/193=46%	1.4s	0.2s	0.3s	0.1s	0.8s
		2	43	37462	44/72=61%	331/563=58%	8.8s	1.8s	1.8s	0.5s	9.0s
		3	905	1895267	69/84=82%	624/661=94%	5.9m	3.0m	1.0m	2.8s	2.4m
StackLi	pop	1	2	196	4/4=100%	25/25=100%	0.2s	0.0s	0.1s	0.0s	0.8s
		2	3	387	4/4=100%	25/25=100%	0.7s	0.0s	0.1s	0.4s	0.3s
		3	4	675	4/4=100%	25/25=100%	0.8s	0.0s	0.1s	0.0s	0.4s
	push	1	4	758	1/1=100%	10/10=100%	1.0s	0.0s	0.2s	0.0s	0.8s
		2	6	1390	1/1=100%	10/10=100%	1.4s	0.0s	0.1s	0.0s	1.4s
		3	8	2260	1/1=100%	10/10=100%	1.9s	0.0s	0.2s	0.0s	1.5s
Vector	add	1	6	818	8/8=100%	77/77=100%	1.5s	0.6s	0.1s	0.0s	1.2s
		2	14	1514	8/8=100%	77/77=100%	4.0s	2.7s	0.3s	0.0s	2.0s
		3	30	2906	8/8=100%	77/77=100%	7.1s	4.3s	0.9s	0.2s	4.6s
	ensureCapacity	1	13	831	8/8=100%	62/62=100%	2.4s	1.1s	0.2s	0.1s	2.2s
		2	29	1703	8/8=100%	62/62=100%	5.7s	2.0s	0.8s	0.9s	4.4s
		3	61	3447	8/8=100%	62/62=100%	9.5s	2.8s	3.3s	0.8s	10.3s
	indexOf	1	6	588	10/10=100%	41/41=100%	2.8s	0.3s	0.3s	0.9s	0.8s
		2	16	1135	10/10=100%	41/41=100%	2.7s	1.0s	0.5s	0.2s	3.0s
		3	38	2339	10/10=100%	41/41=100%	6.3s	1.7s	1.0s	1.2s	8.1s
	insertElementAt	1	6	1131	11/12=91%	105/112=93%	2.0s	0.8s	0.2s	0.1s	1.2s
		2	26	3175	11/12=91%	106/112=94%	6.2s	3.3s	0.7s	0.5s	4.1s
		3	78	9235	11/12=91%	106/112=94%	18.1s	12.6s	2.0s	0.3s	12.2s
	lastIndexOf	1	8	608	12/12=100%	48/48=100%	0.9s	0.2s	0.1s	0.0s	1.1s
		2	18	1141	12/12=100%	48/48=100%	2.2s	0.5s	0.5s	0.2s	3.2s
		3	40	2313	12/12=100%	48/48=100%	5.1s	1.9s	1.1s	0.3s	7.1s
	removeElementAt	1	3	200	7/8=87%	70/71=98%	0.6s	0.1s	0.1s	0.0s	0.4s
		2	5	320	8/8=100%	71/71=100%	0.6s	0.2s	0.1s	0.0s	1.0s
		3	9	566	8/8=100%	71/71=100%	1.4s	0.6s	0.2s	0.1s	1.7s

Table 9.6: *Lazier Experiment Data (3); s – seconds; m – minutes*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_S^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
ABS	abs	1	2	60	2/2=100%	8/8=100%	0.6s	0.1s	0.3s	0.0s	0.3s
AvlTree	find	1	5	3271	10/10=100%	60/60=100%	1.5s	0.4s	0.3s	0.0s	1.4s
		2	29	48244	10/10=100%	60/60=100%	11.5s	2.8s	2.4s	0.2s	4.2s
		3	275	10944306	10/10=100%	60/60=100%	2.0h	1.1h	17.4m	2.1s	59.3s
	findMax	1	2	1457	7/8=87%	38/42=90%	1.6s	0.7s	0.3s	0.1s	0.2s
		2	5	12738	8/8=100%	41/42=97%	3.7s	0.6s	0.4s	0.0s	1.2s
		3	20	2395408	8/8=100%	41/42=97%	8.2m	5.5m	12.5s	1.1s	3.6s
	findMin	1	2	1457	7/8=87%	38/42=90%	1.3s	0.0s	0.3s	0.2s	0.5s
		2	5	12738	8/8=100%	41/42=97%	5.0s	1.9s	0.4s	0.0s	1.7s
		3	20	2395408	8/8=100%	41/42=97%	8.3m	5.6m	14.1s	0.8s	2.9s
	insert	1	5	4719	12/18=66%	119/160=74%	3.0s	0.3s	0.5s	0.4s	1.1s
		2	29	56832	18/18=100%	270/270=100%	12.3s	4.5s	1.8s	0.2s	7.3s
		3	275	11036507	18/18=100%	270/270=100%	2.4h	1.4h	18.3m	5.5s	58.5s
DoubleLinkedList	addBefore	1	8	6178	1/1=100%	31/31=100%	2.8s	0.6s	0.2s	0.1s	1.6s
		2	24	34918	1/1=100%	31/31=100%	9.7s	4.2s	0.8s	0.4s	3.6s
		3	40	99182	1/1=100%	31/31=100%	15.8s	7.5s	0.7s	0.2s	6.9s
	clear	1	1	226	1/1=100%	19/19=100%	0.3s	0.0s	0.1s	0.0s	0.1s
		2	3	1194	1/1=100%	19/19=100%	1.0s	0.1s	0.1s	0.0s	0.4s
		3	4	2798	1/1=100%	19/19=100%	1.5s	0.2s	0.1s	0.0s	0.9s
	indexOf	1	2	668	4/10=40%	28/49=57%	0.6s	0.0s	0.1s	0.0s	0.2s
		2	19	5214	10/10=100%	49/49=100%	3.3s	0.6s	0.4s	0.3s	4.1s
		3	51	15709	10/10=100%	49/49=100%	12.0s	3.4s	3.2s	1.9s	7.4s
	lastIndexOf	1	2	670	4/10=40%	29/50=58%	0.8s	0.4s	0.1s	0.0s	0.4s
		2	19	5214	10/10=100%	50/50=100%	2.8s	0.6s	0.7s	0.1s	4.5s
		3	51	15693	10/10=100%	50/50=100%	9.4s	2.2s	3.4s	0.3s	8.6s
	remove	1	2	664	4/10=40%	26/53=49%	0.8s	0.1s	0.1s	0.0s	0.2s
		2	19	5105	11/12=91%	81/86=94%	4.3s	0.3s	0.5s	0.2s	2.8s
		3	51	15422	11/12=91%	81/86=94%	11.4s	1.5s	4.3s	0.3s	8.5s
	removeLast	1	2	278	1/2=50%	21/46=45%	0.6s	0.1s	0.1s	0.0s	0.0s
		2	6	1677	2/2=100%	46/46=100%	1.1s	0.1s	0.1s	0.0s	1.0s
		3	8	3759	2/2=100%	46/46=100%	2.8s	1.1s	0.3s	0.0s	1.5s
	toArray	1	1	165	1/2=50%	18/27=66%	0.2s	0.0s	0.1s	0.0s	0.1s
		2	30	1973	2/2=100%	27/27=100%	4.9s	2.0s	1.0s	0.7s	5.8s
		3	351	14484	2/2=100%	27/27=100%	24.6s	6.4s	3.8s	1.6s	1.6m
GC	Mark	1	306	226510	12/12=100%	64/64=100%	46.2s	2.1s	12.0s	2.0s	59.6s
TC	classify	1	15	404	16/16=100%	54/54=100%	1.7s	0.6s	0.8s	0.0s	2.2s
StackAr	pop	1	2	105	4/4=100%	32/32=100%	0.4s	0.0s	0.1s	0.1s	0.3s
		2	2	105	4/4=100%	32/32=100%	0.3s	0.0s	0.1s	0.0s	0.1s
		3	2	105	4/4=100%	32/32=100%	0.5s	0.1s	0.3s	0.0s	0.4s
	push	1	4	250	4/4=100%	36/36=100%	0.6s	0.4s	0.1s	0.0s	0.5s
		2	4	250	4/4=100%	36/36=100%	0.7s	0.1s	0.1s	0.1s	0.8s
		3	4	250	4/4=100%	36/36=100%	0.8s	0.1s	0.1s	0.1s	0.4s
AP	partition	1	1	152	1/10=10%	18/68=26%	0.3s	0.0s	0.1s	0.0s	0.1s
		2	4	533	9/10=90%	50/68=73%	1.0s	0.4s	0.1s	0.0s	0.8s
		3	15	2340	10/10=100%	68/68=100%	4.0s	1.4s	0.4s	0.1s	2.6s
BinaryHeap	deleteMin	1	2	290	6/14=42%	71/120=59%	0.5s	0.1s	0.1s	0.0s	0.4s
		2	4	670	6/14=42%	71/120=59%	1.1s	0.4s	0.1s	0.1s	0.4s
		3	11	2327	10/14=71%	101/120=84%	4.0s	2.0s	0.4s	0.3s	1.9s
		4	37	9816	13/14=92%	117/120=97%	13.5s	8.7s	1.5s	0.1s	5.0s
	insert	1	4	458	8/8=100%	65/69=94%	1.0s	0.5s	0.1s	0.0s	0.5s
		2	12	1470	8/8=100%	69/69=100%	2.6s	0.9s	0.5s	0.0s	1.1s
		3	34	5388	8/8=100%	69/69=100%	7.3s	3.2s	1.3s	0.1s	3.1s
		4	113	22909	8/8=100%	69/69=100%	29.9s	19.9s	3.0s	1.2s	9.0s
Sort	insertionSort	1	2	190	1/6=16%	9/44=20%	0.4s	0.1s	0.1s	0.0s	0.3s
		2	5	506	6/6=100%	44/44=100%	1.1s	0.3s	0.1s	0.0s	1.0s
		3	19	2427	6/6=100%	44/44=100%	5.3s	3.0s	1.0s	0.3s	2.8s
	shellsort	1	2	192	1/8=12%	10/61=16%	0.4s	0.1s	0.1s	0.0s	0.5s
		2	5	553	8/8=100%	61/61=100%	1.6s	0.6s	0.3s	0.0s	0.8s
		3	19	2674	8/8=100%	61/61=100%	6.7s	3.3s	1.1s	0.1s	3.2s

Table 9.7: *Lazy Experiment Data (1); s – seconds; m – minutes; h – hours*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_S^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
BinarySearchTree	find	1	13	4890	8/8=100%	65/65=100%	3.2s	0.3s	1.0s	0.2s	2.0s
		2	126	89819	8/8=100%	65/65=100%	29.4s	10.9s	5.8s	0.5s	20.8s
		3	2873	3822839	8/8=100%	65/65=100%	1.8h	42.5m	50.2m	16.8s	9.5m
	findMax	1	3	1430	5/6=83%	29/32=90%	1.1s	0.1s	0.1s	0.0s	0.4s
		2	13	14851	6/6=100%	32/32=100%	5.0s	1.6s	0.6s	0.1s	2.3s
		3	131	331374	6/6=100%	32/32=100%	3.5m	3.0m	8.4s	1.0s	16.9s
	findMin	1	3	1428	5/6=83%	30/37=81%	1.3s	0.1s	0.5s	0.1s	1.1s
		2	13	14888	6/6=100%	37/37=100%	4.8s	1.7s	0.4s	0.1s	3.1s
		3	131	332455	6/6=100%	37/37=100%	3.8m	3.1m	9.5s	2.0s	16.3s
	insert	1	13	6097	6/6=100%	63/63=100%	3.9s	0.9s	0.3s	0.0s	2.1s
		2	112	91691	6/6=100%	63/63=100%	27.4s	9.5s	4.0s	0.7s	20.2s
		3	2161	3349343	6/6=100%	63/63=100%	1.4h	39.1m	34.2m	16.3s	6.7m
	remove	1	13	4146	8/12=66%	68/95=71%	4.4s	1.2s	0.7s	1.2s	1.4s
		2	112	74896	14/16=87%	104/113=92%	26.5s	12.8s	3.7s	0.4s	20.4s
		3	2161	3031511	15/16=93%	111/113=98%	1.2h	35.1m	27.2m	15.0s	5.9m
LL	merge	1	1	1007	3/10=30%	29/76=38%	0.9s	0.0s	0.1s	0.0s	0.2s
		2	6	4707	10/10=100%	76/76=100%	3.1s	0.6s	0.4s	0.3s	3.6s
		3	63	63932	10/10=100%	76/76=100%	16.3s	3.5s	3.0s	0.4s	13.5s
TreeMap	get	1	8	4309	9/10=90%	60/67=89%	4.8s	3.0s	0.6s	0.0s	1.5s
		2	62	85601	9/10=90%	60/67=89%	21.4s	8.5s	4.6s	0.8s	11.1s
		3	782	20707094	9/10=90%	60/67=89%	8.8h	5.0h	1.8h	13.0s	3.6m
	lastKey	1	2	1219	5/6=83%	32/35=91%	0.9s	0.1s	0.1s	0.0s	0.5s
		2	6	15680	6/6=100%	35/35=100%	8.4s	3.5s	0.5s	0.2s	0.9s
		3	31	3524450	6/6=100%	35/35=100%	27.7m	21.5m	40.5s	0.2s	4.3s
	put	1	16	9951	12/32=37%	133/371=35%	4.6s	1.1s	1.0s	0.2s	3.2s
		2	144	181493	40/52=76%	470/500=94%	43.8s	18.7s	8.2s	1.0s	27.5s
		3	N/A	N/A	N/A	N/A	>24h	N/A	N/A	N/A	N/A
	remove	1	7	2247	13/34=38%	105/216=48%	1.8s	0.1s	0.3s	0.1s	1.4s
		2	73	74892	45/74=60%	347/586=59%	20.6s	7.9s	4.3s	0.2s	12.8s
		3	1075	17631620	70/86=81%	640/684=93%	7.5h	3.7h	2.3h	14.3s	4.7m
DisjSetsFast	Find	1	1	282	1/2=50%	14/28=50%	0.7s	0.1s	0.1s	0.0s	0.0s
		2	7	1269	2/2=100%	28/28=100%	2.7s	1.5s	0.3s	0.0s	1.1s
		3	55	6486	2/2=100%	28/28=100%	13.9s	7.6s	2.5s	0.9s	6.2s
	union	2	6	1644	6/6=100%	62/62=100%	2.7s	1.6s	0.3s	0.1s	1.1s
		3	60	10097	6/6=100%	62/62=100%	16.7s	11.1s	1.4s	0.5s	9.0s
AP(I)	partition	1	1	104	3/10=30%	27/62=43%	0.2s	0.0s	0.1s	0.0s	0.1s
		2	3	253	9/10=90%	44/62=70%	0.7s	0.3s	0.2s	0.0s	0.6s
		3	7	692	10/10=100%	62/62=100%	1.6s	0.4s	0.1s	0.0s	0.7s
BinaryHeap(I)	deleteMin	1	2	247	6/14=42%	71/114=62%	0.5s	0.1s	0.1s	0.0s	0.6s
		2	3	407	6/14=42%	71/114=62%	1.1s	0.2s	0.1s	0.1s	0.3s
		3	5	728	10/14=71%	98/114=85%	1.3s	0.5s	0.2s	0.1s	0.8s
		4	8	1288	13/14=92%	111/114=97%	2.2s	1.5s	0.2s	0.0s	1.5s
	insert	1	2	206	6/6=100%	49/53=92%	0.5s	0.1s	0.1s	0.0s	0.5s
		2	5	480	6/6=100%	53/53=100%	1.5s	0.7s	0.2s	0.2s	0.3s
		3	8	837	6/6=100%	53/53=100%	1.9s	0.8s	0.3s	0.0s	0.9s
		4	12	1447	6/6=100%	53/53=100%	2.8s	0.9s	0.5s	0.3s	1.5s
LL(I)	merge	1	1	755	3/10=30%	26/69=37%	0.8s	0.4s	0.1s	0.0s	0.3s
		2	5	3211	10/10=100%	69/69=100%	1.9s	0.1s	0.6s	0.0s	2.2s
		3	19	13689	10/10=100%	69/69=100%	6.6s	1.6s	1.1s	0.4s	3.5s
Sort(I)	insertionSort	1	1	88	3/6=50%	25/41=60%	0.5s	0.3s	0.1s	0.0s	0.0s
		2	3	218	6/6=100%	41/41=100%	0.5s	0.2s	0.2s	0.0s	0.4s
		3	9	770	6/6=100%	41/41=100%	1.5s	0.8s	0.2s	0.0s	1.6s
	shellsort	1	1	98	4/8=50%	35/58=60%	0.7s	0.1s	0.3s	0.0s	0.3s
		2	3	246	8/8=100%	58/58=100%	1.3s	1.0s	0.1s	0.0s	0.4s
		3	9	828	8/8=100%	58/58=100%	2.8s	2.0s	0.3s	0.0s	1.0s
DisjSets	Find	1	1	267	1/2=50%	14/23=60%	0.4s	0.1s	0.1s	0.0s	0.2s
		2	7	1224	2/2=100%	23/23=100%	2.0s	0.6s	0.3s	0.3s	0.8s
		3	55	6302	2/2=100%	23/23=100%	9.0s	5.2s	1.0s	0.1s	9.7s
	union	2	2	1295	2/2=100%	23/23=100%	1.5s	0.7s	0.1s	0.1s	0.4s
		3	20	6939	2/2=100%	23/23=100%	8.6s	5.5s	0.9s	0.1s	2.9s

Table 9.8: *Lazy Experiment Data (2); s – seconds; m – minutes; h – hours*

Class	Method	k	Test Cases Generated	States	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_S^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
AvlTree(I)	find	1	4	1555	10/10=100%	54/54=100%	1.2s	0.1s	0.2s	0.0s	0.8s
		2	21	15118	10/10=100%	54/54=100%	5.7s	1.7s	0.7s	0.1s	3.3s
		3	190	270115	10/10=100%	54/54=100%	53.9s	18.1s	12.6s	0.5s	29.3s
	findMax	1	2	1050	4/4=100%	27/27=100%	0.7s	0.0s	0.1s	0.0s	0.7s
		2	5	6907	4/4=100%	27/27=100%	3.3s	1.3s	0.5s	0.0s	0.4s
		3	20	85427	4/4=100%	27/27=100%	15.5s	8.3s	1.1s	0.5s	2.4s
	findMin	1	2	1050	7/8=87%	38/42=90%	0.6s	0.0s	0.1s	0.0s	0.5s
		2	5	6907	8/8=100%	41/42=97%	2.1s	0.4s	0.2s	0.0s	1.5s
		3	20	85427	8/8=100%	41/42=97%	14.8s	6.8s	1.1s	0.1s	2.0s
	insert	1	4	2460	12/18=66%	113/148=76%	2.9s	0.0s	2.2s	0.0s	0.6s
		2	21	20699	18/18=100%	234/234=100%	7.4s	3.4s	0.8s	0.1s	3.9s
		3	190	329125	18/18=100%	234/234=100%	55.7s	18.8s	10.9s	0.8s	34.9s
BinarySearchTree(I)	find	1	4	1165	1/1=100%	12/12=100%	1.2s	0.0s	0.3s	0.1s	0.8s
		2	21	11037	1/1=100%	12/12=100%	4.9s	1.1s	0.9s	0.1s	3.5s
		3	236	221356	1/1=100%	12/12=100%	50.1s	16.9s	8.7s	0.8s	43.9s
	findMax	1	2	722	3/4=75%	21/24=87%	0.5s	0.0s	0.1s	0.1s	0.4s
		2	5	4453	4/4=100%	24/24=100%	2.3s	0.3s	0.2s	0.0s	0.5s
		3	26	53369	4/4=100%	24/24=100%	11.1s	5.3s	0.9s	0.1s	4.3s
	findMin	1	2	721	3/4=75%	22/29=75%	0.7s	0.0s	0.1s	0.0s	0.4s
		2	5	4463	4/4=100%	29/29=100%	2.0s	0.5s	0.2s	0.0s	0.8s
		3	26	53554	4/4=100%	29/29=100%	11.8s	5.2s	1.0s	0.1s	4.0s
	insert	1	4	1637	6/6=100%	53/53=100%	1.1s	0.2s	0.2s	0.2s	0.7s
		2	21	13194	6/6=100%	53/53=100%	4.5s	0.7s	0.7s	0.1s	3.8s
		3	236	244100	6/6=100%	53/53=100%	54.5s	20.5s	8.7s	0.7s	41.4s
	remove	1	4	1005	8/12=66%	58/85=68%	1.0s	0.4s	0.1s	0.0s	0.5s
		2	21	9845	14/16=87%	94/103=91%	4.6s	0.7s	0.9s	0.3s	5.3s
		3	236	206601	15/16=93%	101/103=98%	41.3s	14.2s	9.0s	0.9s	51.8s
TreeMap(I)	get	1	6	769	8/8=100%	44/44=100%	3.1s	2.5s	0.2s	0.0s	1.1s
		2	71	9310	8/8=100%	44/44=100%	7.4s	2.9s	2.1s	0.3s	17.9s
		3	3863	577272	8/8=100%	44/44=100%	8.7m	3.0m	4.4m	15.4s	18.9m
	lastKey	1	2	615	5/6=83%	32/35=91%	0.8s	0.0s	0.4s	0.0s	0.0s
		2	6	7800	6/6=100%	35/35=100%	3.6s	1.3s	0.3s	0.7s	0.9s
		3	42	305573	6/6=100%	35/35=100%	43.4s	27.7s	1.6s	0.3s	5.3s
	put	1	13	7287	15/34=44%	135/366=36%	3.1s	0.8s	0.3s	0.1s	2.5s
		2	153	146035	43/54=79%	472/495=95%	27.1s	8.0s	5.4s	0.5s	36.9s
		3	5650	10876344	45/54=83%	480/495=96%	41.1m	22.2m	7.1m	17.0s	15.6m
	remove	1	6	1699	12/32=37%	89/193=46%	1.5s	0.5s	0.2s	0.2s	1.2s
		2	121	96554	44/72=61%	331/563=58%	23.0s	6.0s	6.5s	0.3s	25.5s
		3	4495	7900852	69/84=82%	624/661=94%	28.4m	14.3m	5.4m	11.1s	13.2m
StackLi	pop	1	2	196	4/4=100%	25/25=100%	0.2s	0.0s	0.0s	0.0s	0.3s
		2	3	425	4/4=100%	25/25=100%	0.7s	0.0s	0.1s	0.2s	0.4s
		3	4	770	4/4=100%	25/25=100%	1.0s	0.0s	0.4s	0.0s	0.6s
	push	1	4	758	1/1=100%	10/10=100%	1.0s	0.0s	0.2s	0.1s	1.0s
		2	6	1466	1/1=100%	10/10=100%	1.3s	0.0s	0.2s	0.2s	1.2s
		3	8	2450	1/1=100%	10/10=100%	2.3s	0.4s	0.2s	0.0s	1.3s
Vector	add	1	6	986	8/8=100%	77/77=100%	2.4s	1.0s	0.3s	0.0s	1.8s
		2	20	2932	8/8=100%	77/77=100%	7.4s	4.2s	0.6s	0.3s	3.0s
		3	74	10990	8/8=100%	77/77=100%	22.9s	15.8s	1.5s	0.4s	9.7s
	ensureCapacity	1	17	1051	8/8=100%	62/62=100%	3.3s	1.1s	0.5s	0.5s	3.8s
		2	57	3239	8/8=100%	62/62=100%	6.6s	2.3s	0.9s	0.1s	12.9s
		3	205	11339	8/8=100%	62/62=100%	20.1s	9.5s	5.4s	0.5s	46.4s
	indexOf	1	7	644	10/10=100%	41/41=100%	1.2s	0.2s	0.2s	0.1s	1.4s
		2	17	1195	10/10=100%	41/41=100%	3.2s	0.5s	0.5s	0.6s	3.3s
		3	44	2686	10/10=100%	41/41=100%	6.0s	2.5s	1.1s	0.4s	9.3s
	insertElementAt	1	6	1425	11/12=91%	105/112=93%	3.6s	2.0s	0.1s	0.0s	1.0s
		2	41	6874	11/12=91%	106/112=94%	14.4s	10.1s	1.1s	0.2s	5.5s
		3	210	41077	11/12=91%	106/112=94%	1.6m	1.3m	6.9s	0.5s	27.3s
	lastIndexOf	1	9	662	12/12=100%	48/48=100%	0.8s	0.2s	0.1s	0.1s	1.3s
		2	19	1199	12/12=100%	48/48=100%	2.8s	0.8s	0.8s	0.2s	4.1s
		3	46	2650	12/12=100%	48/48=100%	8.5s	3.6s	1.1s	0.6s	5.6s
	removeElementAt	1	3	202	7/8=87%	70/71=98%	0.9s	0.6s	0.1s	0.0s	0.2s
		2	6	382	8/8=100%	71/71=100%	1.4s	0.2s	0.4s	0.0s	0.7s
		3	16	999	8/8=100%	71/71=100%	3.5s	0.7s	1.1s	0.5s	3.5s

Table 9.9: *Lazy Experiment Data (3); s – seconds; m – minutes; h – hours*

For each lazy/lazier/lazier# result table, the columns in the table report on the size of k, number of test cases generated, bytecode-level branch and instruction coverage, total running time (excluding time for generating object graph visualizations) followed by a breakdown of this time into time spent in the CVC Lite theorem prover [11], time to run the concretization algorithm (reverse execution $\Rightarrow^{-1}$ and POOC [57]), and time to form the JUnit tests. Finally, the last column gives the additional time not included in the total time column to generate the object graph visualizations. In general, lazier# is better than lazier which in turn is better than lazy in terms of smaller numbers of states, smaller numbers of cases, and shorter total running/theorem prover times. However, there are some anomalies in running time and theorem prover time comparison. For example, all AvlTree methods with $k = 3$, the lazier# takes more total running time and theorem prover time than lazier. This is because (for reasons unknown to us) the underlying theorem prover, CVC Lite [11], takes more time under the lazier# initialization algorithm for this example (even though lazier# invokes the theorem prover fewer times). If we examine the difference between total time and theorem prover time which is actual running time of the algorithms, it follows the general trend: lazier# is shorter than lazier.

We will discuss the important columns such as number of test cases, coverage, timing as follows:

- **Number of test cases** Given our case analysis in Chapter 4, we are most interested in the number of cases explored. We have three observations about the numbers of cases:
  Observation 1: for some examples, such as AVL tree, the numbers of cases are the same for lazier and lazier#. This is because in the example, NULL is not allowed for tree elements and this confirms our previous observation that lazier initialization is optimal for non-NULL data.
  Observation 2: for lazier#, the numbers of cases of binary search tree, AVL tree, and red-black tree insertion match exactly with the numbers calculated by the combinatorics technique discussed in Chapter 4 – thus establishing that the algorithm is case-optimal for these examples (which are the most complicated ones in our example pool).
  Observation 3: all the numbers of cases for search/insert/remove operations are the same

for each tree (binary search tree, AVL tree, red-black tree) under the lazier# algorithm. This is because the search/insert/remove operations that involve finding the position for the element first and rest operations (inserting or removing tree node and then re-balance the tree) are deterministic. So the calculations for insertion are applicable to the search and remove operations.

- **Coverage Levels:** As expected, the cost of analysis and number of test cases grow exponentially as k increases. Thus, we were encouraged (and somewhat surprised) to find out that a value of $k = 2$ was sufficient for achieving 100% branch coverage in almost all cases, and $k = 3$ was required to get the highest levels of coverage for only the `treeMap` and `BinaryHeap` examples. We have confirmed and documented in Section 9.2 that almost all cases in which 100% branch coverage is not achieved represent pathological cases of infeasible branches/dead code. Only for `TreeMap` did we encounter feasible branches (2 for `put`, 3 for `remove`) that were not covered at $k = 3$.

- **Computation Cost:** The `java.util.TreeMap` example is the most complex from our study. The two most complicated methods of this example required 2.2 minutes in both cases for $k = 3$. The total running time for a vast majority of the methods is under a few seconds. When considering future work, it is important to note that typically one half to two thirds of the time is spent in the theorem prover. There are significant opportunities for optimizations (e.g., caching results across theorem proving calls), but these will require us to work with the developers of CVC Lite to design a new collection of top-level APIs.

- **Size of Test Suite:** As noted earlier, the data clearly indicates exponential growth in the number of tests as k increases. In most cases, the running time to generate all tests for a k-bound that yields 100% feasible branch coverage is under 10 seconds.

  Note that our numbers of tests are what results from considering all heap configurations within the k bound – not the number of tests required to achieve 100% branch coverage. In some situations, test cases beyond what are necessary to achieve 100% branch coverage are generated. It is relatively easy to modify Kiasan/KUnit so that it avoids exploring a path after branch coverage has been achieved. We are implementing this modification for the purpose of measuring the number of test cases beyond those required for 100% branch coverage.

- **Annotation Burden:** The size of the required invariant methods is relatively small compared to the overall size of the classes. For instance, in our most complicated example (`TreeMap`), the invariant is 74 non-comment source statements (NCSS) while the total NCSS is 414 (invariant is 17.87% of total). Across all of our examples, the invariant is typically 10-18% of the entire class NCSS. It is worth noting that from a single invariant, one is able to obtain a huge benefit in the form of automatically generated test suites. We believe the disparity between the effort required to write an invariant and the effort required to write tests manually (which iteratively trying to achieve a coverage) is so great as to render negligible any complaints about the effort required to provide the invariant annotations.

9.2 TreeMap Coverage

The class java.util.TreeMap from Java 1.5 library is a red-black tree implementation. We tested two most important methods from the class: `put` (inserting an element into the tree) and `remove` (deleting an element from the tree) using KUnit. For $k = 3$, KUnit reports a branch coverage of 40/52 for the `put` method and 70/86 for the `remove` method.

So for testing of the `remove` method, there are 16 uncovered branches. We organize the uncovered branches according to the feasibility as follows.

- Inherent infeasible paths, 6: one path in each of `setColor, parentOf, rightOf`, and `leftOf` methods and two paths in `deleteEntry` method. The `setColor` method shown in Figure 9.1 has a test of null-ness of the argument at line 2 which is always true. The `setColor` method is private and only called from `fixAfterInsertion`, shown in Figure 9.3, and `fixAfterDeletion`, shown in Figure 9.4, methods. The calling contexts always guarantee that the argument is non-null. The same is for the `parentOf, rightOf`, and `leftOf` methods.

For method `deleteEntry` shown in Figure 9.5, the conditional at line 39 is alway true because line 33 has already done the test; similarly, the conditional at line 42 is always true because of the conditional at line 40.

- Infeasible paths due to specification, 1: the `compare` method tests the null-ness of the field `comparator` at line 5 of Figure 9.2. And we put `comparator!=null` in the specification for both methods.

- Infeasible paths due to context, 6. All 6 paths are in `successor` method shown in Figure 9.6. In our testing of `remove`, the `successor` method is only called from `deleteEntry` at line 8 with an argument that has both left and right children. So as shown in Figure 9.6, the conditional at lines 5 is alway false and the conditional at line 7 is alway true. Furthermore, lines 13-29 are unreachable code which contains 4 branches. So there are total 6 infeasible branches due to context.

- Feasible paths, 3: one in each of `rotateRight, rotateLeft`, and `deleteEntry` methods.

For testing of the `put` method, there are 10 uncovered branches. We organize the uncovered branches according to the feasibility as follows.

- Inherent infeasible paths, 7: one path in each of `setColor, parentOf, rightOf`, and `leftOf` methods and three paths in `fixAfterInsertion` method. The reason for the infeasible paths in `setColor, parentOf, rightOf`, and `leftOf` methods is the same as explained above.

  The three infeasible paths in `fixAfterInsertion` shown in Figure 9.3 are more involved and related to the loop invariant of the main loop in the method body.

Method `fixAfterInsertion` is a helper function that is called from `put` after a node is inserted in the red-black tree. The inserted node `x` is colored RED as shown in line 2. The goal of this method is to adjust the tree after insertion to satisfy the red-black tree invariants. The only red-black tree invariant that the new tree could violate is "all children of a RED node have to be BLACK" because the parent node of the inserted node could be RED too. KUnit reports (and we confirmed) that this method contains three infeasible paths related to the loop invariant. The main loop from line 4 to line 40 deals with the case that node `x` is RED and its parent is also RED. Part of the loop invariant is that `x!=null` and the color of `x` is RED. To see that is the case, first, `x` is non-NULL and RED before entering the loop; second, inside the loop, `x` is either unchanged or moved up the tree to the grandparent of `x` which can not be NULL because the parent of `x` is RED and the root of the tree is BLACK and further `x` is colored with RED. Thus, `x!=null` at line 4 and the conditional at line 19 are always true. Symmetrically, the conditional at line 36 is always true.

- Infeasible paths due to specification, 1: the `compare` method tests the null-ness of field comparator and we put "comparator!=null" in the specification for both methods.
- Feasible paths, 2: one in each of `rotateRight` and `rotateLeft` methods.

The uncovered branches in `rotateRight` and `rotateLeft` methods are different in testings of `remove` and `put` methods. In summary, for all the feasible branches, there is only one in `deleteEntry` not covered by KUnit.

```
    private static <K, V> void setColor(Entry<K, V> p, boolean c) {
        if (p != null)
            p.color = c;
    }
```

Figure 9.1: *Method setColor*

```
    /**
     * Compares two keys using the correct comparison method for this TreeMap.
     */
    private int compare(K k1, K k2) {
        return (comparator == null ? ((Comparable</*-*/K>) k1).compareTo(k2)
                : comparator.compare((K) k1, (K) k2));
    }
```

Figure 9.2: *Method compare*

```
    private void fixAfterInsertion(Entry<K, V> x) {
        x.color = RED;

        while (x != null && x != root && x.parent.color == RED) {
            if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
                Entry<K, V> y = rightOf(parentOf(parentOf(x)));
                if (colorOf(y) == RED) {
                    setColor(parentOf(x), BLACK);
                    setColor(y, BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    x = parentOf(parentOf(x));
                } else {
                    if (x == rightOf(parentOf(x))) {
                        x = parentOf(x);
                        rotateLeft(x);
                    }
                    setColor(parentOf(x), BLACK); //bug seeded
                    setColor(parentOf(parentOf(x)), RED);
                    if (parentOf(parentOf(x)) != null)
                        rotateRight(parentOf(parentOf(x)));
                }
            } else {
                Entry<K, V> y = leftOf(parentOf(parentOf(x)));
                if (colorOf(y) == RED) {
                    setColor(parentOf(x), BLACK);
                    setColor(y, BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    x = parentOf(parentOf(x));
                } else {
                    if (x == leftOf(parentOf(x))) {
                        x = parentOf(x);
                        rotateRight(x);
                    }
                    setColor(parentOf(x), BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    if (parentOf(parentOf(x)) != null)
                        rotateLeft(parentOf(parentOf(x)));
                }
            }
        }
        root.color = BLACK;
    }
```

Figure 9.3: *Method fixAfterInsertion*

```
private void fixAfterDeletion(Entry<K, V> x) {
    while (x != root && colorOf(x) == BLACK) {
        if (x == leftOf(parentOf(x))) {
            Entry<K, V> sib = rightOf(parentOf(x));
            if (colorOf(sib) == RED) {
                setColor(sib, BLACK);
                setColor(parentOf(x), RED);
                rotateLeft(parentOf(x));
                sib = rightOf(parentOf(x));
            }
            if (colorOf(leftOf(sib)) == BLACK
                    && colorOf(rightOf(sib)) == BLACK) {
                setColor(sib, RED);
                x = parentOf(x);
            } else {
                if (colorOf(rightOf(sib)) == BLACK) {
                    setColor(leftOf(sib), BLACK);
                    setColor(sib, RED);
                    rotateRight(sib);
                    sib = rightOf(parentOf(x));
                }
                setColor(sib, colorOf(parentOf(x)));
                setColor(parentOf(x), BLACK);
                setColor(rightOf(sib), BLACK);
                rotateLeft(parentOf(x));
                x = root;
            }
        } else { // symmetric
            Entry<K, V> sib = leftOf(parentOf(x));
            if (colorOf(sib) == RED) {
                setColor(sib, BLACK);
                setColor(parentOf(x), RED);
                rotateRight(parentOf(x));
                sib = leftOf(parentOf(x));
            }
            if (colorOf(rightOf(sib)) == BLACK
                    && colorOf(leftOf(sib)) == BLACK) {
                setColor(sib, RED);
                x = parentOf(x);
            } else {
                if (colorOf(leftOf(sib)) == BLACK) {
                    setColor(rightOf(sib), BLACK);
                    setColor(sib, RED);
                    rotateLeft(sib);
                    sib = leftOf(parentOf(x));
                }
                setColor(sib, colorOf(parentOf(x)));
                setColor(parentOf(x), BLACK);
                setColor(leftOf(sib), BLACK);
                rotateRight(parentOf(x));
                x = root;
            }
        }
    }
    setColor(x, BLACK);
}
```

Figure 9.4: *Method fixAfterDeletion*

```
private void deleteEntry(
Entry<K, V> p) {
    decrementSize();

    // If strictly internal, copy successor's element to p and then make p
    // point to successor.
    if (p.left != null && p.right != null) {
        Entry<K, V> s = successor(p);
        p.key = s.key;
        p.value = s.value;
        p = s;
    } // p has 2 children

    // Start fixup at replacement node, if it exists.
    Entry<K, V> replacement = (p.left != null ? p.left : p.right);

    if (replacement != null) {
        // Link replacement to parent
        replacement.parent = p.parent;
        if (p.parent == null)
            root = replacement;
        else if (p == p.parent.left)
            p.parent.left = replacement;
        else
            p.parent.right = replacement;

        // Null out links so they are OK to use by fixAfterDeletion.
        p.left = p.right = p.parent = null;

        // Fix replacement
        if (p.color == BLACK)
            fixAfterDeletion(replacement);
    } else if (p.parent == null) { // return if we are the only node.
        root = null;
    } else { // No children. Use self as phantom replacement and unlink.
        if (p.color == BLACK)
            fixAfterDeletion(p);

        if (p.parent != null) {
            if (p == p.parent.left)
                p.parent.left = null;
            else if (p == p.parent.right)
                p.parent.right = null;
            p.parent = null;
        }
    }
}
```

Figure 9.5: *Method deleteEntry*

Class	Method	*d*	Time	Branch Coverage	Paths Iterations	Test Cases
AvlTree	find	10	5.9s	10/10=100%	738	13
	findMax	10	15s	8/8=100%	30	11
	findMin	10	14s	4/4=100%	30	11
	insert	27	1h 1m	12/18=67%	6848	-
BinaryHeap	deleteMin	X	X	X	X	X
	insert	X	X	X	X	X
BinarySearchTree	insert	8	7s	6/6=100%	18	9
	remove	17	2.5m	15/16=100%	321	11
DisjSets	find	X	X	X	X	X
	union	X	X	X	X	X
DisjSetsFast	find	X	X	X	X	X
	union	X	X	X	X	X
DoubleLinkedList	addBefore	600	1h 4m	0/1=0%	2064	-
	indexOf	1000	1h 24m	0/10=0%	2593	-
	remove	1000	1h 34m	0/10=0%	2593	-
	clear	1000	58m	0/1=0%	1199	-
	lastIndexOf	1000	1h 34m	0/10=0%	2596	-
	toArray	1000	57m	0/2=0%	1199	-
java.util.TreeMap	lastKey	25	1h 22m	5/6=83%	9077	-
	remove	25	1h 23m	16/86=19%	9079	-
	put	25	1h 22m	15/52=29%	9079	-
java.util.Vector	add	X	X	X	X	X
	ensureCapacity	X	X	X	X	X
	indexOf	X	X	X	X	X
	insertElementAt	X	X	X	X	X
	lastIndexOf	X	X	X	X	X
	removeElementAt	X	X	X	X	X
Stack Array	push	X	X	X	X	X
	pop	X	X	X	X	X
Stack List	push	1000	1h 26m	0/1=0%	3902	-
	pop	3	2s	2/2=100%	2	2
Sort	insertionSort	X	X	X	X	X
	shellsort	X	X	X	X	X
GC	Mark	19	51s	12/12=100%	107	10
IntLinkedList	merge	29	32s	10/10=100%	17	15
ABS	abs	3	3s	2/2=100%	5	3
TriangleClassification	classify	5	5s	3/16=19%	8	4
ArrayPartition	partition	X	X	X	X	X

Table 9.10: *jCUTE Data*

```
/**
 * Returns the successor of the specified Entry, or null if no such.
 */
private Entry<K, V> successor(Entry<K, V> t) {
    if (t == null)
        return null;
    else if (t.right != null) {
        Entry<K, V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    } else {
        Entry<K, V> p = t.parent;
        Entry<K, V> ch = t;
        while (p != null && ch == p.right) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

Figure 9.6: *Method successor*

9.3 Comparison to jCUTE [59]

9.3.1 Experiment Methodology

Recall that a theme of the Kiasan approach is controlling the scope and cost of the analysis via the k-bound on the length of reference chains. In contrast, jCUTE aims to achieve as high degree of branch coverage as possible with the cost and scope of the analysis controlled by a depth bound. One interesting point of the comparison between the Kiasan and jCUTE is to consider the amount of branch coverage (and the performance in achieving that coverage) that the respective tools can achieve on different examples. In the Kiasan examples, we highlighted the bound value required for achieving 100% branch coverage, and in cases where 100% coverage could not be achieved, we reported the coverage achieved.

We conduct jCUTE experiments in an analogous way driven by the desire to maximize coverage. In particular, we try to find the minimal depth bound required to achieve 100% branch coverage. In cases where 100% is not achieved, we attempt to determine that amount of coverage that can be achieved in roughly one hour. In jCUTE it is possible to constraint the number of "iterations" of the algorithm (equivalent to the number of paths explored). For simplicity, we do not constrain the number of iterations, that is, we set the depth and let jCUTE run to finish. The experiment data is recorded if any of following conditions is true:

- reached desired branch coverage (100% coverage of feasible branches);
- depth reached 1000;
- run time more than one hour;
- jCUTE error occurred (as the TriangleClassification example).

Table 9.10 displays the results of the experiments. jCUTE currently does not handle arrays of variable size (situations where the size of array is not known statically). This means that jCUTE can not process a number of the examples used in our experiments. These situations are indicated in Table 9.10 with the symbol *X*.

9.3.2 Summary of jCUTE Performance

1. **AVL Tree:** jCUTE performance is comparable to Kiasan on methods find, findMax, and findMin. For each of these cases, Kiasan results are actually better, but still within the same order of magnitude. For example, for find, Kiasan achieves 100% branch coverage on $k = 1$ in 1.8s (compared to 5.9s for jCUTE) and generates 4 tests instead of 13. For findMax, Kiasan achieves 100% branch coverage on $k = 2$ in 2.7s (compared to 15s for jCUTE) and generates 5 tests instead of 11.

   However, the performance diverges significantly for the insert method. jCUTE obtains only 12/18 branch coverage (compared to 18/18 for Kiasan), and runs for over one hour (compared to 8.9s for $k = 2$ for Kiasan – the point at which 18/18 coverage was reached).

2. **Binary Heap**, jCUTE can not handle this example with dynamically sized arrays so no data is reported.

3. **Binary Search Tree**, jCUTE was executed for methods insert and remove. jCUTE performance is comparable to Kiasan on all methods. Kiasan is almost better on all the data. For

example, for insert, Kiasan takes 2.1s (compared to 7s for jCUTE) to reach 100% branch coverage with $k = 1$ and generates 6 (compared to 9 for jCUTE) test cases. For remove, Kiasan achieves 15/16 branch coverage in 1.5m (compared to 2.5m for jCUTE) with $k = 3$ and generates 236 (compared to 11 for jCUTE) test cases. However, jCUTE explores 321 paths to achieve this coverage.

4. **Disjoint Set (Original/Fast)**, jCUTE can not handle these two examples with dynamically sized arrays so no data is reported.

5. **Double Linked List** is taken from java.util.LinkedList. jCUTE executed for methods addBefore, indexOf, remove, clear, and lastIndexOf. In all the example, Kiasan does much better than jCUTE: jCUTE spent more than or about one hour for each method but the coverage is 0%. We are currently corresponding with jCUTE creators to determine why jCUTE is not able to make any progress on this example.

6. **TreeMap**, jCUTE executed for methods lastKey, remove, and put. Kiasan performs significantly better than jCUTE. For example, for remove, Kiasan achieves 69/84 branch coverage in 5m 51s with $k = 3$ while jCUTE takes more than one hour to reach 16/84 coverage. The other two method comparisons are similar.

7. **Vector**, jCUTE can not handle this example with dynamically sized arrays so no data is reported.

8. **Stack Array Implementation**, jCUTE can not handle this example with dynamically sized arrays so no data is reported.

9. **Stack List Implementation**, jCUTE executed for methods push and pop. For pop, jCUTE's performance is comparable with Kiasan. Kiasan achieves 2/2 branch coverage with 0.5s (compared to 2s for jCUTE) with $k = 1$ and generates 2 (the same as jCUTE does) test cases.

However, for push, jCUTE performs much worse: after more than one hour without covering any of the tested code while Kiasan uses 1.4s to achieve 100% branch coverage with $k = 1$.

10. **Sort**, jCUTE can not handle this example with dynamically sized arrays so no data is reported.

11. **GC**, jCUTE executed for method Mark. jCUTE performs better than Kiasan but still within the same order of magnitude. jCUTE achieves 100% branch coverage with 51s (compared to 1m 51.9s for Kiasan) and generates 10 (compared to 306 for Kiasan) test cases.

12. **IntLinkedList**, jCUTE executed for method merge. Kiasan and jCUTE are comparable in merge method. Kiasan takes 12.4s to achieve 100% branch coverage (compared to 32s for jCUTE) with $k = 3$ and generates 30 (compared to 15 for jCUTE) test cases.

13. **ABS**, jCUTE executed for method **abs**. jCUTE and Kiasan performance are similar. Kiasan is actually better but within same magnitude. Kiasan achieves 100% branch coverage in 0.5s (compared to 3s for jCUTE) and generates 2 (compared to 3 for jCUTE) test cases.

14. **TriangleClassification**, jCUTE executed for method classify. jCUTE can not handle this example and throws 4 internal errors. All the errors are the same and the error message is listed as follows:

```
Error in Thread[main,5,main] null
java.lang.ArrayIndexOutOfBoundsException: 1
at cute.concolic.a.g.a(ArithmeticExpression.java:91)
at cute.concolic.a.g.a(ArithmeticExpression.java:128)
at cute.concolic.b.b.a(ComputationStack.java:273)
at cute.concolic.b.c.a(ComputationStacks.java:94)
at cute.concolic.Call.branchPos(Call.java:299)
at TriangleClassification.classify(TriangleClassification.java:42)
at TriangleClassification.main(TriangleClassification.java:59)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:585)
at cute.RunOnce.main(RunOnce.java:242)
```

15. **Array Partition**, jCUTE can not handle this example with dynamically sized arrays so no data is reported.

9.4 Comparison to JPF [37]

Direct comparison to JPF is non-trivial for several reasons. First, to run JPF's symbolic execution, one must hand-instrument programs to incorporate the symbolic execution functionality which makes large case studies like the ones that we have carried out infeasible. (Recent work by Anand et. al. [5] provides an automated transformation for JPF, but that tool component has not been released yet.) Moreover, JPF does not actually generate test cases at this point (it only emits information about symbolic states). To enable a direct comparison between the underlying algorithmic strategies in both tools, we have implemented the "lazy initialization" approach of JPF originally introduced in [37]. Having corresponded closely with NASA Ames personnel, we are confident that our implementation of lazy initialization reflects the strategy implemented in JPF. Given the underlying lazy initialization algorithm, we then had to decide which bounding strategy to use: should we choose the JPF depth-bounded strategy or KUnit's *k*-bounded strategy? Because one of our primary objectives is to demonstrate the benefits of our lazier# initialization approach over lazy initialization, we wanted to focus on exactly that factor in the experiments. Thus, we implemented a *k*-bounding strategy for lazy initialization (removing the additional *k*-bounding/depth-bounding factor) and the results are shown in Tables 9.7, 9.8, and 9.9. In general, lazier# produces significantly smaller state-spaces (and thus significantly smaller test suites as reflected in the Test Cases column). In the most complex examples, lazier# gives several orders of magnitude reduction in time to generate test suites, e.g., reducing time required to achieve the highest levels of coverage from over 24 hours down to 2.2 minutes for `TreeMap.put`. The reductions grow exponentially as *k* increases.

Chapter 10

Related Work

In this chapter, we present related work organized by the following categories: model checking (Section 10.1), abstract interpretation (Section 10.2), enumerative methods (Section 10.3), deductive methods (Section 10.4), symbolic execution (Section 10.5), and traditional testing (Section 10.6). Note that many tools and approaches are combinations of two or more techniques; we discuss each tool under its main technique. The classification is certainly biased and only reflects the author's view.

10.1 Model Checking

Model checking [17] is a technique that given a model M of a system and a property P to be checked, it systematically enumerates all the states of M to check whether P is satisfied. Model checking has its initial successes in checking hardware systems. Then it is applied to software specification and protocols [35]. Recently, model checking is applied to Java: JPF [15] and Bandera [19]. The initial version of Bandera translates Java source code into multiple input languages of backend model checkers such as Spin [35]. The new Bandera [54] uses Bogor as its backend. JPF has a customized Java VM which can model check Java bytecode directly.

The advantages of model checking are:

- it can check deep semantic properties such as temporal properties;
- it can reason about multithreaded programs and detect problems due to intricate interleav-

ings of threads;

- it naturally produces error traces.

However, due to its exhaustive nature, it has following disadvantages:

- it requires huge space to store visited states because the number of states usually is at least exponential with respect to the size of the grogram; The problem is being referred to as the *state explosion* problem.

- it requires closed systems to get finite models, M; For open systems, user has to close the systems first in order to apply miodel checking.

- the model M may not have exact the same behavior as the system; if M is an over-approximation of the system, it may report spurious errors; if M is an under-approximation of the system and no error is found, it is not conclusive.

For model checking systems in different domains, new model checkers such as Bogor [54] have been built with extensibility. In Bogor, new language constructs can be added as first-class language constructs to extend the input language, and core modules can be customized to leverage domain specific behaviors. Bogor has been used to model check a real-time event channel of avionics systems [22] and publisher/subscriber systems [9] recently. Following the successes of Bogor, XRT [32], an extensible model checking framework for .NET systems, is introduced by Microsoft Research.

Kiasan is built on top of the Bogor model checking framework. It can be viewed as a stateless (no state is stored) search technique. Compared with the stateful (visited states are stored) model checking, Kiasan may take more time because it may visit a state multiple times while stateful model checking avoids the revisiting by storing the visited states. Thus, Kiasan trades off space against time.

10.2 Abstract Interpretation

Abstract interpretation [20] is a foundational framework for deriving and proving correctness of static analysis. While many analyses including model checking and symbolic execution[1] can be viewed as abstract interpretation, we only discuss scalar data abstraction (SLAM [7]) and heap abstraction (TVLA [42]) in this section.

SLAM [7] The scalar data abstraction used in SLAM is *predicate abstraction* which abstracts all the scalar data into boolean predicates. For example, three integer variables x, y, z in a program may be abstracted into two predicates $x > y$ and $x + y < z$. Then the program is conservatively transformed into a boolean program that only manipulates the boolean variables which correspond to the boolean predicates. Thus, the state-space of the program is greatly reduced. Similarly to Kiasan, SLAM is a forward and path-sensitive analysis based on a model checking engine. It can verify temporal safety properties and simple assertions in sequential C programs. SLAM has been successfully applied to Windows device drivers. It computes an over-approximate behavior of the program and uses *iterative refinement* to discover more predicates. Although the whole process is automatic, termination is not guaranteed. In contrast, Kiasan computes an under-approximation and termination is guaranteed by the k-bound. Furthermore, SLAM is not targeted towards strong properties that Kiasan is designed for.

TVLA [42] TVLA is a shape analyzer which can verify programs that manipulate the heap. It is a flow sensitive but not path-sensitive analysis. In contrast, Kiasan is a path-sensitive (and flow sensitive) analysis. TVLA uses a technique called heap abstraction which abstracts a potentially infinite heap graph into a finite one where sets of nodes that are indistinguishable by the properties are grouped into summary nodes. More specifically, TVLA works at the representation of first order logic with transitive closure. It uses an extension of 2-value logic where 0 represents false and 1 represents true, 3-value logic with the introduction of 1/2 (unknown) to represent sum-

[1]In fact, the soundness and completeness proofs of Kiasan are based on abstract interpretation.

mary nodes. The lazy/lazier/lazier# initialization algorithms used in Kiasan can also be viewed as heap abstraction techniques where the symbolic objects/locations/references are the summary nodes. Similarly to lazy initialization, TVLA materializes a summary node, which is called *focus* in TVLA, when performing operations on it. To terminate the analysis, TVLA includes a *blur* [42] operation which merges heap shapes after the focus operation whereas the lazy/lazier/lazier# initialization algorithms never do. Thus, the lazy/lazier/lazier# initialization is more precise but at the cost of non-termination. And Kiasan has to use *k*-bound (and sometime loop bounding) to guarantee termination.

The main advantage of TVLA is that it can verify heap manipulating programs because it computes an over-approximation of the state. However, the over-approximation may introduce false alarms. The main disadvantage of TVLA is that, in general, user is required to provide *instrumentation predicates* to facilitate the verification. However, there is some progress [52, 44] towards automatic instrumentation predicate discovery. On the contrary, Kiasan trades full soundness with automation. That is, Kiasan is unsound (under-approximation) due to the *k*-bound but is fully automatic. User can gain more confidence by using larger *k* which will have higher computation cost. Future work on Kiasan includes incorporating new developments in heap abstraction and making Kiasan more expressive without sacrificing the automation.

10.3 Enumerative Methods

In this section, we discuss several tools that limit the scope of analysis.

Alloy [36] uses a specification language that consists of first order logic with relations and has an automatic analyzer which is based on boolean satisfiability (SAT) solving to generate instances that satisfy the specification. Since in general, the first order logic is undecidable, Alloy bounds the scope of the analysis to make the analysis decidable. The strategy that Alloy used to limit the scope is very similar to Kiasan's *k*-bound technique. It also introduces unsoundness because there may be errors that need larger bounds to be exposed. That is, Alloy is also an under-approximation.

Similarly to Kiasan, the bound can be adjusted by the user and increased to gain higher levels of confidence by using more computation resources. In contrast to Kiasan, Alloy is not lazy by setting the size of universe, thus may generate larger instances than necessary. Another difference is that Alloy's input language is first order logic with relations which may be more difficult for programmers to write than the Java specification and code in Kiasan.

TestEra [45] uses Alloy to test Java programs, specifically, to generate test inputs using preconditions and check postconditions. The translations from Alloy to Java (for test inputs) and Java to Alloy (for end states) are done manually.

TestEra uses a similar idea as Kiasan in checking programs, that is, it *assume*s the precondition to generate inputs, runs the program, and *assert*s the postcondition. However Kiasan does all those three steps in Java and is fully automatic while TestEra involves manual translations between Java source code and the Alloy input language.

Korat [14] can generate non-isomorphic heap structures for black-box testing in a given bound. Similarly to Kiasan/KUnit, Korat works at the level of Java source code and JML specification. Although we have only presented KUnit in generating white-box test cases, KUnit can also be used to generate black-box test cases by just symbolically executing the precondition. Korat uses a *finitization* technique which in essence bounds the number of nodes in the heap compared to Kiasan's longest-reference-chain bound. Korat generates all possible inputs and then uses the precondition to prune illegal ones. It observes that when some fields of the precondition are not accessed, the non-accessed parts are irrelevant for the partial structures; and incorporates the knowledge as an optimization. This optimization can be seen as a crude approximation of lazy initialization. Furthermore, Korat focuses on generating heap structures only but not primitive data in the structures.

10.4 Deductive Methods

Most deductive method approaches are based on Hoare logic [34]. In this section, we give an overview of representative tools and approaches of deductive methods: ESC/Java, LOOP, Separation Logic, KeY, and PALE.

ESC/Java [26, 18, 10] is an automatic tool that supports lightweight JML contracts and can check open systems. Kiasan differs from ESC/Java in two important aspects: analysis technique and heap representation.

For analysis technique, ESC/Java uses the weakest precondition calculus which is a backward and flow sensitive but path-insensitive analysis. One problem with such approach is that it is difficult to generate counterexamples for contract violations (i.e., test cases illustrating violations). Recent work [21] tries to address this issue by processing ESC/Java failed proof attempts, and then running programs with random inputs to check whether the warnings are false alarms (if not, it found a test case). This seems to work well for scalar data, however it does not work with heap intensive programs and contracts (since ESC/Java itself targets at lightweight properties). In contrast, since Kiasan uses a forward and path-sensitive analysis, KUnit can generate a concrete input and a JUnit test case for each error (such as postcondition violations, uncaught exceptions). We believe that Kiasan/KUnit provides an alternative solution for contract-based static checking that is able to reason about strong heap-oriented properties, while the work presented in this thesis takes us further in term of analysis feedback compared to [21].

For heap representation, ESC/Java uses a logical representation of the heap. Due to limitations of the underlying theorem prover on reasoning about the heap graph, ESC/Java cannot handle strong heap properties. Kiasan maintains an explicit representation of the heap structure and logical relations only for primitive data which allow Kiasan to check strong properties. However, Kiasan's explicit representation incurs a heavy cost on enumerating all possible aliasings. Further study is still needed to compare the different representations of the heap.

LOOP [61] can also verify JML specification in Java source code in PVS [50]. Compared to Kiasan, LOOP also has a rigorous foundation: it has formalized the JML and Java semantics in PVS. The semantics is based on Hoare logic and the weakest precondition calculus and has been proved sound in PVS. LOOP translates JML specification together with Java source code into PVS [50] proof obligations which are Hoare triples. There are two main differences between LOOP and Kiasan: automation and representation level that user works on. First, the proving of translated PVS obligations in LOOP is manual, thus, labor intensive. In contrast, Kiasan is fully automatic but cannot do full verification – it does only bounded verification. Second, LOOP user has to prove the obligations in the PVS level which requires a lot of PVS expertise. Although Kiasan works at JVM bytecode level, user only needs to write code and specification in Java because Kiasan is automatic and the analysis result is given as graphs and JUnit test.

Separation Logic [53] is an extension of Hoare logic to reason about programs that manipulate heap. To concisely specify heap structures, it introduces two operators, *separation conjunction* and *separation implication*, and associated rules to reason about them. Separation conjunction is used to indicate that sub-formulae hold in separated parts of heap and likewise separation implication. The initial analyses in using separation logic are all manual. Recent development, Smallfoot [13], makes an initial step to automate separation logic by using a symbolic execution technique.

KeY [12] uses a dynamic logic and symbolic execution to verify programs in Java Card, a strict subset of Java language for smart cards and embedded systems. The specification language can be Object Constraint Language (OCL) or JML. KeY translates specification and Java source code into proof obligations in the dynamic logic which is an extension of Hoare logic. The semantics of OCL and Java Card is formalized in the dynamic logic. Similar to Kiasan, KeY is a forward and path-sensitive analysis. However, it differs in two important aspects from Kiasan: state representation and automation. KeY has a pure logical representation of the state in contrast to Kiasan's hybrid representation: heap is concrete and the path condition which constraints the primitive symbols is logical. KeY is not fully automatic: it requires user to provide loop invariants whereas

Kiasan uses *k*-bound to achieve full automation without requiring loop invariants (which are ofter difficult to obtain for object-oriented programs) at the price of introducing unsoundness.

PALE [48] is a framework to verify program behaviors that manipulate the heap. It translates a given program annotated by graph types into weak monadic second-order logic formulae, then uses the underlying decision procedure MONA [33] to verify them. The graph types are heap structures with tree backbones. To achieve automation, PALE requires the user to annotate loop invariants. Contrary to Kiasan, PALE is sound (an over-approximation). However PALE cannot reason about the integer type. Thus PALE cannot express any properties that involving integers, for example, a part of red-black tree property: all paths from the root to a leaf node contain the same number of black nodes.

10.5 Symbolic Execution

Besides JPF symbolic execution[37, 62], there are other tools and approaches using symbolic execution: CUTE, Unit Meister, and Symstra.

JPF [37, 62] We have compared Kiasan/KUnit approach with JPF approach in many places of this thesis. Here we just give a summary of the differences in two parts. The first part is the comparison between Kiasan with JPF symbolic execution[37]:

- Kiasan introduces two improved core algorithms for handling objects in symbolic execution: the lazier and lazier# initialization algorithms which are shown significant improvements over the original lazy initialization algorithm in [37];
- Kiasan can reason about open systems;
- Kiasan supports compositional reasoning;
- Kiasan uses a longest-reference-chain bounding strategy in contrast to to JPF's search depth bound or symbolic object size bound;

- JPF [37] symbolically executes the method body and the precondition is only used for pruning infeasible paths. In order to use the precondition, JPF has to reconstruct the lazily initialized pre-state by using a mapping. Kiasan directly executes precondition first to generate pre-states and then checks the method body. In short, Kiasan only executes the precondition once and does not have to reconstruct the pre-state each time that lazy initialization occurs as JPF does;

- Kiasan is fully automatic while JPF requires manual instrumentation. However, recent work [5] for JPF provides an automatic translation that makes progress towards removing manual instrumentation.

The second part is the comparison between KUnit with JPF test input generation [62]:

- KUnit uses lazier# initialization algorithm which is a significant improvement over the lazy initialization used in [62] in terms of the number of test cases generated; The lazier# algorithm has been rigorously demonstrated to generate the optimal numbers of test cases for several complex data structures.

- KUnit uses modified backtracking rules for symbolic execution to generate test inputs;

- KUnit's input generation algorithm has been formalized and proved sound;

- KUnit can generate interface mock objects for open systems which are not considered in the JPF work;

- KUnit generates visualization of heap graphs and reports coverage metrics.

CUTE There is an interesting line of work starting with [29] that uses simultaneous symbolic and concrete executions focusing on non-heap-intensive C programs. A recent extension [28] provides a compositional approach in which method behaviors are summarized as pre/postconditions. Building on [29], CUTE [59] is a branch coverage driven approach for unit testing. More specifically, CUTE uses concrete execution for testing and symbolic execution to guide concrete execution to cover branches. KUnit maintains a precise visible part of heap while CUTE uses constraints to maintain the relations of heap nodes. KUnit has a theoretical proof of its path coverage modulo *k*-bound while CUTE may miss paths due to incomplete consideration of aliasings. Finally, KUnit can generate mock objects for open systems. We believe the approaches are complementary; that is, Kiasan/KUnit's systematic exploration of all heap configurations under a bound is suitable for heap intensive programs, and the concolic approach can be used to handle complex arithmetic constraints and native library calls (although this causes unsoundness). Further study is needed to compare the use of pre/postconditions in our work with that of [28].

Unit Meister [60] Unit Meister from Microsoft can also generate test cases from Parameterized unit tests (PUTs) for .NET assemblies. PUTs are just wrappers of the methods to be tested and their specifications. Based on symbolic execution, Unit Meister does a forward and path-sensitive analysis as Kiasan. However, Unit Meister's symbolic states are different from Kiasan's in four ways. First, it includes function symbols for compositional checking whereas Kiasan only allows primitive symbols and compositional checking is done by using the specification. Second, it uses an algebraic representation of the heap in contrast to Kiasan's concrete heap representation. Third, while Kiasan is fully automatic, depending on the theorem prover and constraint solver, Unit Meister may require user to supply domains for heap nodes. Last difference is the bounding strategy: Unit Meister uses a loop bound compared with Kiasan's preference on the bounding based on longest-reference-chain .

Symstra [65] is another symbolic execution tool for generating minimal sequences of public method calls to test a class. Symstra uses symbolic primitive values, concrete heap structures, and state subsumption to generate non-isomorphic end states. In KUnit, primitive data and heap structures are all symbolic. Furthermore, KUnit does not have state subsumption. The reason that it is not done in Kiasan/KUnit is that it is hard to compare heaps with symbolic nodes. Another difference is that KUnit generates pre-states by preconditions and invariants; while Symstra uses sequences of public method calls. Thus Symstra generates a subset of possible valid pre-states.

Depending on the quality of preconditions and invariants, KUnit may generate a superset/subset of all possible valid pre-states. However, since KUnit gives useful feedback, user can modify preconditions or invariants to make them more precise.

10.6 Testing

Traditional testing [49] is the most common used technique for software quality assurance. The advantages of testing are:

- flexible, it can be used to test any property as long as user can write a test case;
- low barrier to entry, it does not require a lot of training because most programmers are familiar with the practice.

The disadvantages of testing are:

- inconclusive, testing can only find bugs but not verify programs;
- expensive, 50% of development time and cost [49] are spent on testing;
- hard to ensure the quality of test cases since they are written by developers and may contain errors;
- hard to quantify the functional coverage due to lack of functional metrics.

There are two testing strategies: black-box testing and white-box testing. Black-box testing is only based on specification without inspection of the implementation. White-box testing (glass-box in some literatures) allows the examination of the code. In black-box testing, the criterion is *exhaustive input testing*[49] which in general is impossible. So techniques such as equivalence partition (partitioning the input spaces and testing each partition), boundary-value analysis (writing testing input around the boundary values) etc. are used to guide test case writing. In white-box testing, the ultimate goal is to have *exhaustive path coverage*. Due to the infinite number of paths in most programs, branch and statement coverages are proposed to approximate the path coverage.

KUnit can be used for black-box testing in the following way: generating inputs by symbolically executing the precondition and using the postcondition as the oracle. If the formal specification is present, KUnit approach can be seen as a formalized method for black-box testing techniques such as equivalence partition, boundary value analysis by partitioning the input space by the path that the input corresponds to. Furthermore, KUnit guarantees a heap coverage which is a functional metric. For white-box testing, KUnit guarantees the path coverage within the k bound. In connection to traditional coverages such as statement and branch coverage, KUnit gives such coverage reports and our experiment shows that for small k ($k = 2$), almost all our examples reach nearly 100% feasible branch coverage.

In general, traditional testing is based on informal specification while KUnit requires formal specification which can be seen as a drawback. However, considering the amount of effort spent in writing test cases, we believe that KUnit has a right trade-off on benefits and efforts.

Chapter 11

Conclusion

We have presented Kiasan, an alternative technique to reason about open systems based on symbolic execution that is able to check strong heap properties. We have implemented Kiasan on top of the Bogor framework. Methodologically, we envision our tool being used similarly to frameworks like ESC/Java [26]. For example, a user can start checking a method (even compositionally) without any annotation and receive error feedback from the tool.

We have proposed two consecutively improved algorithms, lazier and lazier# initialization, over the original lazy initialization algorithm presented in [37]. We have provided operational semantics for symbolic executions with lazy/lazier/lazier# initialization algorithms and the concrete execution on JVM. Then we have shown that symbolic executions with lazy/lazier/lazier# initializations are relatively sound and complete. We have presented a case counting method to quantify heap coverage for their evaluation. We have demonstrated the lazier initialization algorithm and the early lazy initialization algorithm from [37] are sub-optimal on some complex data structures such as the red-black tree implementation in `java.util.TreeMap`. We have also shown that the lazier# algorithm is optimal on those data structures with respect to the counting method. We have also presented empirical case studies to demonstrate the effectiveness of lazier# initialization algorithm compared to the lazy and the lazier initialization algorithms.

We have presented, KUnit, a framework built on the Kiasan symbolic execution engine for unit test case generation, input/output object graph visualization, and mock object generation. Our efforts have highlighted the concept and utility of guaranteeing complete coverage of heap configurations up to bounds on the length of reference chains in data structures. This strengthens previous testing approaches by guaranteeing full soundness (no errors will be missed) for any data whose size lies within the supplied Kiasan *k*-bound. Using a broad range of heap-intensive examples, we have shown that our approach can achieve 100% feasible branch coverage (improving on related techniques) by using only small heap configurations and with minimal number of generated tests (again, improving on previous approaches by reducing number of tests generated). From another viewpoint, these capabilities add significant values to Kiasan as a contract checking tool because they dramatically improve upon the quality of feedback provided by error traces and counterexample information in other tools for ESC/Java [21].

Bibliography

[1] +http://junit-objects.sourceforge.net/+.

[2] Asm website. |http://asm.objectweb.org/—.

[3] Bogor website. |http://bogor.projects.cis.ksu.edu—, 2003.

[4] A. V. Aho and N. J. A. Sloane. Some doubly exponential sequences. *Fibonacci Quarterly*, 11:429–437, 1970.

[5] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependency analysis and program transformation for symbolic execution. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[6] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstract subsumption checking. *SPIN Workshop on Model Checking of Software*, 2006.

[7] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.

[8] Tomas Ball. A theory of predicate-complete test coverage and generation. Technical report, Microsoft Research, 2004.

[9] Luciano Baresi, Carlo Ghezzi, and Luca Mottola. On accurate automatic verification of publish-subscribe architectures. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 199–208, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, 3362:49–69, 2004.

[11] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. *Computer-aided Verification (CAV)*, pages 515–518, 2004.

[12] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[13] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. *Asian Symposium on Programming Languages and Systems*, pages 52–68, 2005.

[14] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM Press, 2002.

[15] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, Jul 2000.

[16] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language. In *Proceedings of the International Conference on Software Engineering Research and Practice*, 2002.

[17] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, jan 2000.

[18] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 108–128, 2004.

[19] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, jun 2000.

[20] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[21] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering*, pages 422–431. ACM Press, May 2005.

[22] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, and Robby. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects (SOFMFCAB)*, nov 2002.

[23] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A *k*-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 157–166, 2006.

[24] Xianghua Deng, Robby, and John Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *Testing: Academic and Industrial Conference – Practice and Research Techniques*, 2007. To appear.

[25] Xianghua Deng, Robby, and John Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. In *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, 2007.

[26] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[27] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, objects. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion*, pages 236–246. ACM, 2004.

[28] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, volume 42, pages 47–54. ACM Press, 2007.

[29] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM Press, 2005.

[30] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Reading, Massachusetts: Addison-Wesley, 2 edition, 1994.

[31] Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser, second edition, 1982.

[32] Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. XRT - exploring runtime for .NET - architecture and applications. *Workshop on Software Model Checking*, 2005.

[33] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 89–110, 1995.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[35] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.

[36] Daniel Jackson. Alloy: A lightweight object modelling notation.

[37] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for Construction and Analysis of Systems*, pages 553–568, 2003.

[38] Sarfraz Khurshid and Yuk Lai Suen. Generalizing symbolic execution to library classes. In *Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[39] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[40] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, second edition, 1998.

[41] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, October 1998.

[42] T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene-based static analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS)*, 2000.

[43] Tim Lindholm and Frank Yellin. The Java virtual machine specification (2nd edition). http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

[44] Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. Abstraction refinement via inductive learning. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 519–533. Springer, 2005.

[45] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *16th IEEE Conference on Automated Software Engineering (ASE 2001)*, 2001.

[46] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[47] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.

[48] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. *Programming Language Design and Implementation*, pages 221–231, 2001.

[49] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing*. John Wiley & Sons, 2 edition, June 2004.

[50] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (Lecture Notes in Computer Science 607)*, 1992.

[51] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[52] G. Ramalingam, Alex Varshavsky, John Field, Deepak Goyal, and Shmuel Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *PLDI*, pages 83–94, 2002.

[53] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[54] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.

[55] Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking JML specifications using an extensible software model checking framework. *International Journal on Software Tools for Technology Transfer*, 8(3):263–272, 2006.

[56] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill Science/Engineering/Math, 4 edition, December 1998.

[57] Hans Schlenker and Georg Ringwelski. POOC - a platform for object-oriented constraint programming. In B. O'Sullivan, editor, *Recent Advances in Constraints: Joint ERCIM/-CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, pages 159–170. Springer, June 2002.

[58] David Schmidt. Binary relations for abstraction and refinement. Technical report, Kansas State University, November 2000.

[59] Koushik Sen and Gul Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 263–272, 2005.

[60] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. Technical report, Microsoft Research, 2005.

[61] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science 2031)*, 2001.

[62] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation in Java Pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[63] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1998.

[64] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, Inc, 1990.

[65] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.

Appendix A

Kripke Structures

The presentation in this appendix is adapted from [58], and it is provided here for a quick reference.

Definition 11 (Kripke Structure)**.** *A Kripke structure is a quadruple, $\mathcal{K} = (\Sigma_{\mathcal{K}}, I_{\mathcal{K}}, \longrightarrow_{\mathcal{K}}, L_{\mathcal{K}})$, where $\Sigma_{\mathcal{K}}$ is a set of states, $I_{\mathcal{K}}$ is a set of initial states that $I_{\mathcal{K}} \subseteq \Sigma_{\mathcal{K}}$, $\longrightarrow_{\mathcal{K}} \subseteq \Sigma_{\mathcal{K}} \times \Sigma_{\mathcal{K}}$ is the transition relation (finite image), and $L_{\mathcal{K}} : \Sigma_{\mathcal{K}} \to \mathcal{P}(Atom)$ associates a set of atomic properties, $\forall s \in \Sigma_{\mathcal{K}}$ and $L_{\mathcal{K}}(s) \subseteq Atom$.*

Definition 12 (Simulation Relation on Kripke Structures)**.** *For Kripke structures $C = (\Sigma_C, I_C, \longrightarrow_C, L_C)$ and $\mathcal{S} = (\Sigma_{\mathcal{S}}, I_{\mathcal{S}}, \longrightarrow_{\mathcal{S}}, L_{\mathcal{S}})$, a binary relation, $\mathcal{R} \subseteq \Sigma_C \times \Sigma_{\mathcal{S}}$, is a simulation of C by $\mathcal{S}$, written $C \lhd_{\mathcal{R}} \mathcal{S}$, if $\forall c \in \Sigma_C, s \in \Sigma_{\mathcal{S}}. c \mathrel{\mathcal{R}} s \wedge c \longrightarrow c' \implies \exists s' \in \Sigma_{\mathcal{S}}. s \longrightarrow s' \wedge c' \mathrel{\mathcal{R}} s'$ and $\forall c_0 \in I_C. \exists s_0 \in I_{\mathcal{S}}. c_0 \mathrel{\mathcal{R}} s_0$.*

Definition 13 (Left-/Right-total Simulation Relations)**.** *A binary relation, $\mathcal{R} \subseteq S \times T$, is left total if $\forall s \in S. \exists t \in T. s \mathrel{\mathcal{R}} t$. The relation is right total if $\forall t \in T. \exists s \in S. s \mathrel{\mathcal{R}} t$.*

Definition 14 (Power Kripke Structure)**.** *For a Kripke structure, $\mathcal{K} = (\Sigma_{\mathcal{K}}, I_{\mathcal{K}}, \longrightarrow_{\mathcal{K}}, L_{\mathcal{K}})$, the power kripke structure $\mathcal{P}(\mathcal{K}) = (\mathcal{P}(\Sigma_{\mathcal{K}}), \mathcal{P}(I_{\mathcal{K}}), \overset{\bullet}{\longrightarrow}_{\mathcal{K}}, L_{\mathcal{P}(\mathcal{K})})$, where $\forall S, S' \subseteq \Sigma_{\mathcal{K}}. S \overset{\bullet}{\longrightarrow}_{\mathcal{K}} S'$ if and only if for every $s' \in S'$, there exists some $s \in S$ such that $s \longrightarrow_{\mathcal{K}} s'$ and $L_{\mathcal{P}(\mathcal{K})}(S) = \cap \{ L_{\mathcal{K}}(s) \mid s \in S \}$.*

Appendix B

Lazy, Lazier, and Lazier# Swap States

B.1 Lazy Swap States

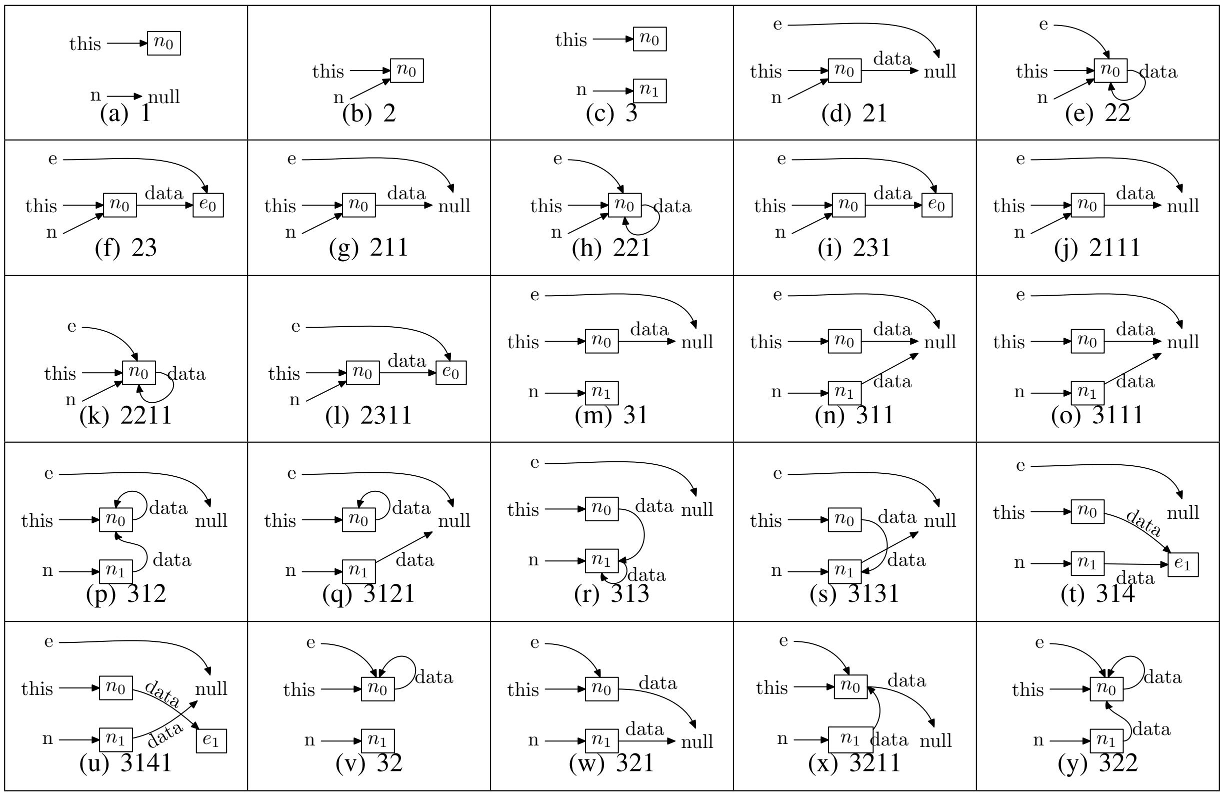
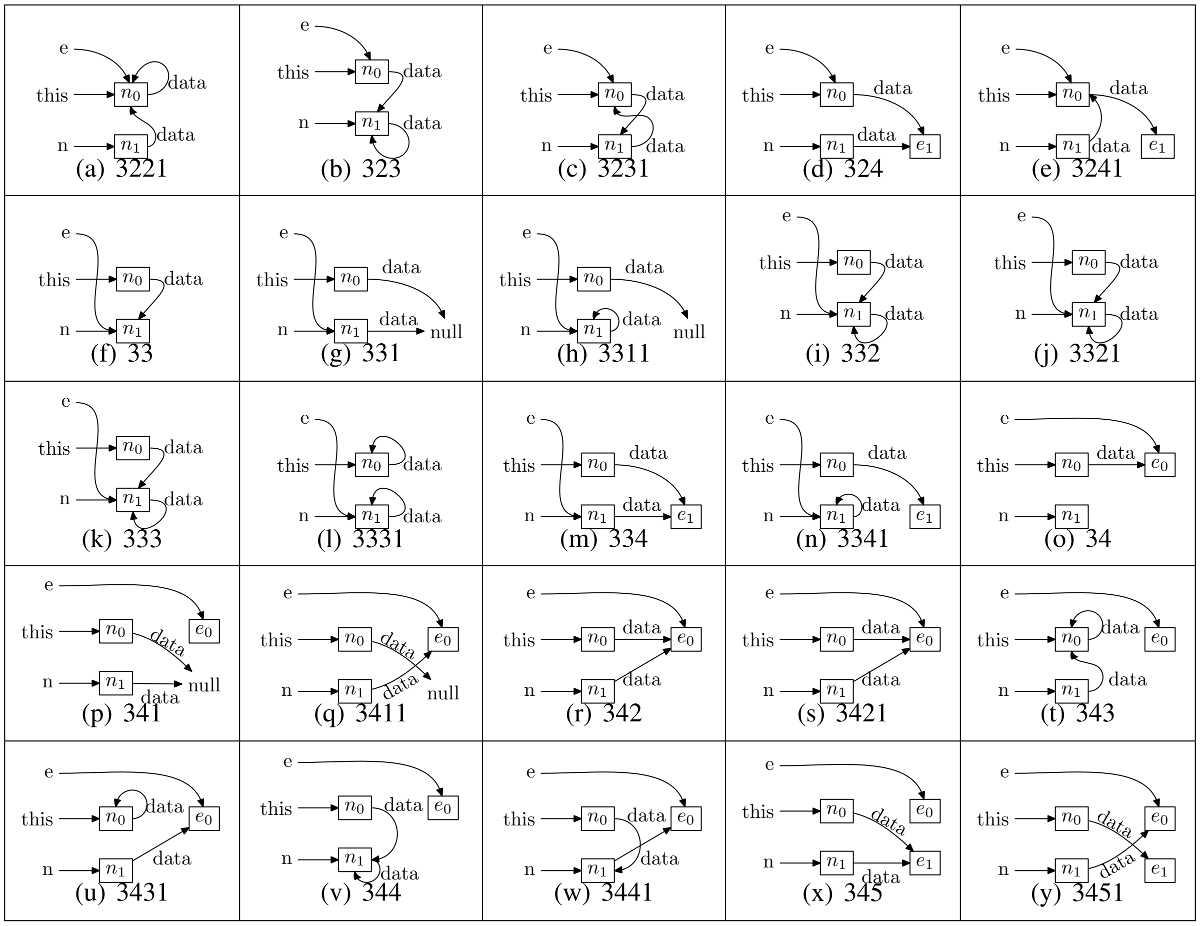


Figure B.1: *Swap–Lazy States (1)*



(a) 3221 (b) 323 (c) 3231 (d) 324 (e) 3241
(f) 33 (g) 331 (h) 3311 (i) 332 (j) 3321
(k) 333 (l) 3331 (m) 334 (n) 3341 (o) 34
(p) 341 (q) 3411 (r) 342 (s) 3421 (t) 343
(u) 3431 (v) 344 (w) 3441 (x) 345 (y) 3451

Figure B.2: *Swap–Lazy States (2)*

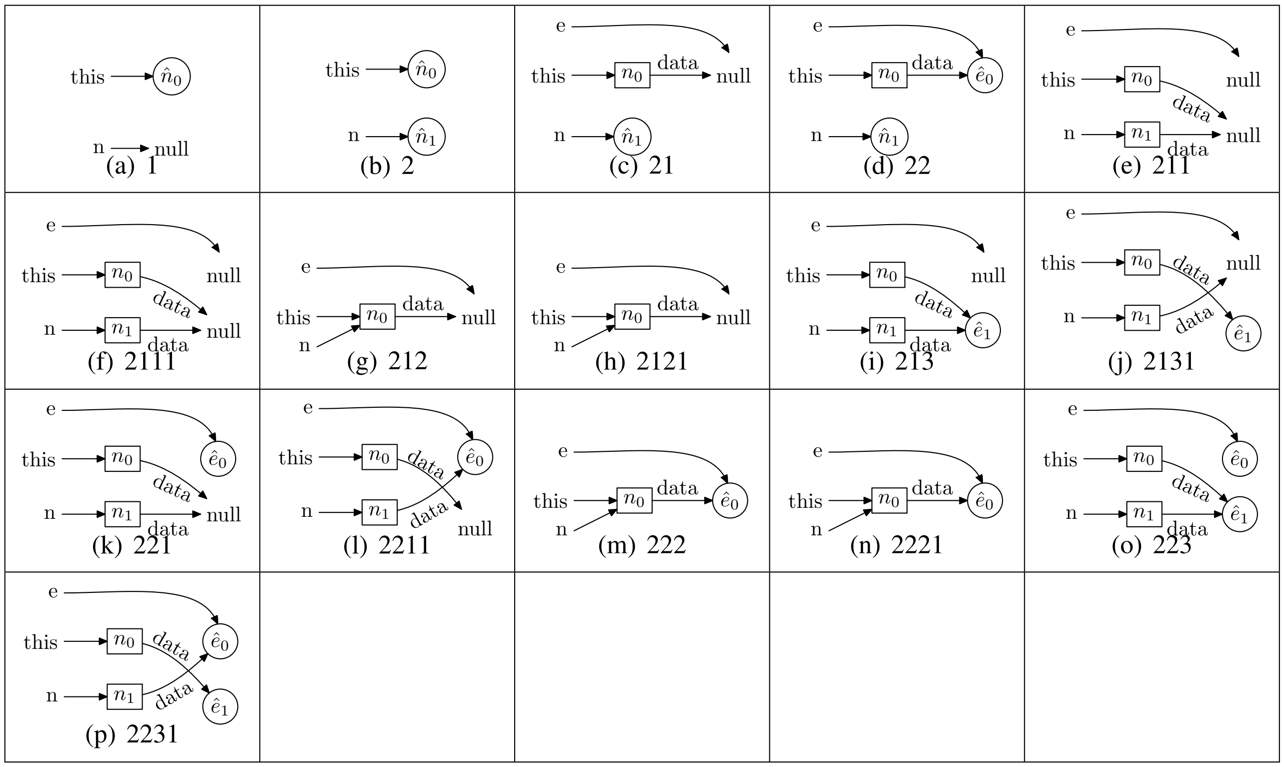


Figure B.3: *Swap–Lazier States*

B.2 Lazier Swap States

B.3 Lazier# States

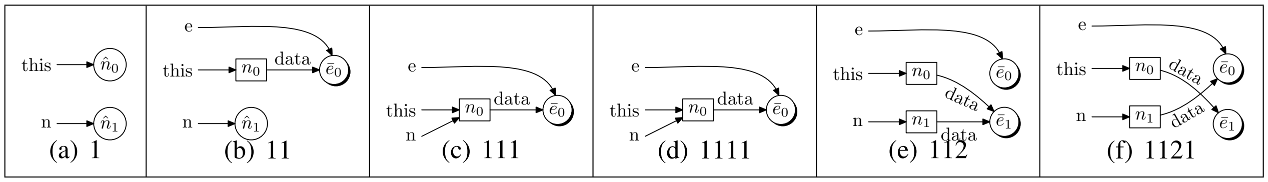


Figure B.4: *Swap–Lazier# States*

Index