EFFICIENT EXPLICIT-STATE MODEL CHECKING FOR PROGRAMS WITH DYNAMICALLY ALLOCATED DATA

BY

MARCELO D'AMORIM

Bach., Universidade Federal de Pernambuco, Brazil, 1997
Mestre, Universidade Federal de Pernambuco, Brazil, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Despite the technological advances in languages and tools to support program development, programmers still deliver software with lots of errors. Software testing has been the dominant approach in industry to improve the quality of software at code level. Testing software is not cheap though. A study done in 2002 by the National Institute of Standards and Technology (NIST) [NIS02] reports that between 70% and 80% of development costs is due to testing. We believe that automation of software testing can help to reduce this cost; automation can assist the programmer to test the code he develops in a more systematic way.

Software model checkers are becoming increasingly popular to assist in the automation of software testing. Model checkers are tools that systematically explore the state space of a model (say, a software model) to demonstrate the presence of errors or to confirm their absence. For example, a model checker can report a test whenever it finds an "interesting" state, i.e., a state which violates a user-defined assertion or a state that makes execution throw a runtime exception.

This dissertation focuses on model checkers that operate directly on code. Model checking programs is becoming very popular. It enables the discovery of errors in the implementation as opposed to errors in a system design.

However, model checking *programs* poses particular challenges. It is the implementation that defines the model, not the design. As a consequence, a deterministic step of the exploration can be potentially more expensive as it may require the execution of a complex fragment of sequential code. In addition, the states that the model defines are potentially larger, making important operations that depend on the size of the state potentially slower.

While state-space explosion has been traditionally the key issue in model checking, *time efficiency* becomes more relevant with the increasing appearance of model checkers that operate on programs. Our goal is to improve time efficiency for model checking programs that allocate dynamic

data. Such programs are pervasive nowadays with the wide use of object-oriented languages.

To that end, this dissertation presents Mixed execution and ΔExecution. Mixed execution speeds up the execution of deterministic steps in model checkers that use a special representation of state. Such representation can be convenient to perform some model checking operations but sacrifices the execution of deterministic steps (i.e., fragments of sequential code). ΔExecution uses sets to perform state-space exploration. The use of sets enables the model checker to take advantage of overlapings (of state and paths) that exist in a systematic exploration of state, speeding up several operations in software model checking.

We implemented mixed execution in JPF. We evaluate mixed execution on seven subject programs and one large case study. The experimental results on the seven smaller programs show that mixed execution can improve the overall time for state exploration in JPF from 1.01x to 1.73x (with median 1.13x). We also evaluate mixed execution on a larger case study containing bugs. The exploration time on this experiment reduces from 1.14x to 1.41x (with median 1.23x).

We implemented Delta Execution in two model checkers, JPF (Java PathFinder) and BOX (*Bounded Object eXploration*) to evaluate the effectiveness of this technique in model checkers with different designs. The results show that ΔExecution improves the overall exploration in both tools, but the improvements are due to different factors. We evaluate ΔExecution on ten simple subject programs and a larger case study with errors. The results show that ΔExecution improves the exploration time for the smaller programs from 0.88x to 126.80x (with median 5.60x) in JPF and from 0.58x to 4.16x (with median 2.23x) in BOX, while taking from 0.46x to 11.50x (with median 1.48x) less memory in JPF and from 0.18x to 2.71x (with median 1.18x) memory in BOX. We also evaluate ΔExecution, using our JPF implementation, on one larger case study with errors, the AODV routing protocol, where ΔExecution improves the exploration time from 0.88x to 2.04x (with median 1.72x).

To Fernanda

Acknowledgments

This project would not have been possible without the support of many. Firstly, I would like to thank my advisor, Prof. Darko Marinov. I owe Darko for his patience and guidance. I became Darko's student on May 2005 after taking an individual study with him. Darko helped me focus on a specific problem that lead to this dissertation and to collaborate with bright graduate students.

I want to thank specially the other members of my thesis committee: Prof. Gul Agha, Prof. Jennifer Hou, and Prof. Mahesh Viswanathan. They offered valuable comments to improve my dissertation and research. I also want to thank specially Prof. Grigore Roşu. I worked with Grigore for almost 2 years in runtime verification, and learned a lot with him. I started working with Darko when realized that my major interest was software testing. I owe Grigore for all the time he dedicated to me and his patience.

I want to thank my colleagues in Siebel Center: Ahmed Sobeih, Amr Ahmed, Andrei Popescu, Azadeh Farzan, Brett Daniel, Choonghwan Lee, Danny Dig, Feng Chen, Karin Straus, Kely Garcia, Koushik Sen, Luiz Ceze, Mark Hills, Steven Lauterburg, Train Serbanuta, and certainly several others. My friends helped to critique my work, and to define how I should approach science.

Thanks to all my friends outside Siebel Center: Aguinaldo Maciente, Austeclinio Farias (Auster), Brett Graham, Bruno Laranjeira, Daniela Mattos, Enlinson Mattos (Castor), Fabio Mattos, Fabio Nectario, Fernando Mesquita, Ivanete Maciente, Juan Carlos Medina, Juan Carlos Paz, Mariano Canteiro, Octavio Sequeiros, Victor Niekel, and possibly many others. You were key in making the stay of my family in Urbana-Champaign a rich life experience.

Thanks to CAPES, the Brazilian funding agency for post graduation, for providing financial support during the first 4 year of my Ph.D. studies.

Finally, I want to thank my wife Fernanda, son Leonardo, my mother Maria Celiomar, my father Claudio, my sisters Adriana and Camila, and my close friends, Carlos Eduardo (Rochinha),

Douglas, Gustavo, Henderson (Jeba), Marcelo, and Orlando, whom I always felt around offering encouragement and love.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

EMC   Explicit-State Model Checking

SMC   Symbolic Model Checking

ΔExecution Delta Execution

ΔState   Delta State

JPF   Java Pathfinder

JVM   Java Virtual Machine

Chapter 1

Introduction

A study done in 2005 by Scaffidi et al. [SSM05] estimates that 90 million of Americans will use computers by 2012. Of these, 13 million will be software programmers. Currently, the number of programmers is between 2 and 2.5 million according to the U.S. Census Bureau [oLS05]. This study not only illustrates the importance of software in the current activities of the American population but also the increasing demand on software development. Society uses software more pervasively nowadays. We can already find software, for example, in cell phones, fridges, cars, or medical equipment, in addition to the personal computer with which people are more familiar.

Despite the technological advances in languages and tools to support program development, programmers still deliver software with lots of errors. A study done in 2002 by the National Institute of Standards and Technology (NIST) [NIS02] estimates that (1) the annual cost of software errors is around 59.5 billion dollars, (2) testing and debugging consume around 70% and 80% of development time, and (3) fixing an error takes on average 3.9h.

The first estimate presents the cost of software errors exposed in various development phases, including later phases when fixing an error becomes more expensive [Boe91]. The second and third estimates point out that testing and debugging take a significant fraction of the development efforts. This study helps in understanding the impact of software errors on the American economy and suggests the need for better techniques to assure software quality. It indicates, in particular, the need for an improved infrastructure for testing.

Testing is the dominant approach in industry for assuring software quality at code level. The principle of testing is simple: a test consists of a sequence of experiments against the *subject* of test and a check, called *test oracle*, that verifies whether the test passes or fails [Bei90]. A test for an *object-oriented program* consists of a sequence of method calls and a procedure that checks whether the execution of the sequence is in accordance to its expected behavior. A *test driver* is a

component that creates environments necessary to test the subject. The driver constructs sequences of method invocations that exercise the subject under different scenarios, effectively exploring its different states.

As the estimates above show, testing and debugging can be very expensive activities in software development. We believe that *automation* of software testing can help to reduce this cost; automation can assist the programmer to test the code he develops in a more systematic way. One way to automate testing is to use model checkers: tools that can explore, in a more systematic fashion, different states of a program.

A *model checker* takes as input a *model* of a system and a *property* about this system and produces as output a description of the behaviors of the system that violate the property. A model includes a set of *states* and a *transition relation* that tells how the system transits from one state to another. *Model checking* is the process of exploring the state space of a model to find property violations or to demonstrate that none exist. *Model checkers* are tools that automate this process. We use the term *state space* to refer to the set of all possible states that the dynamic execution of a test subject can produce. A model checker can, for instance, systematically produce tests with the assistance of a test driver. The tool can report a test whenever it finds an "interesting" state. For example, when the execution of a test reaches a state which violates a user-defined assertion or a state that makes execution throw a runtime exception.

Traditionally, model checkers operate on *design models*, i.e, the input model describes a system design. Examples of such model checkers include FDR [Ros94], SPIN [Hol97], NuSMV [CCGR99], and Alloy Analyser [JSS00]. Finding design flaws is very important: the later a programmer addresses a design error the more costly it is to fix the error.

Recently, however, several model checkers have been implemented to operate on *implementation models*, i.e., the input model describes a system implementation. Examples include JPF [LV01, VHB+03], CMC [MPC+02], BogorVM [RDH03], and SpecExplorer [VCST05]. Finding errors in the implementation is also very important; it allows the programmer to find and fix errors directly in the system's code.

Two important and orthogonal decisions in the design of a model checker are how it represents state and what kind of model it expresses. Chapter 2 discusses these alternatives in more detail.

This dissertation focuses on model checkers that represents state explicitly and take implementation models as input. The goal is to *improve the efficiency of explicit-state model checking for programs that allocate dynamic data.* Improving time efficiency in model checking makes it more practical for the user.

In the following, we discuss the challenges involved in software model checking. We then state the thesis of this dissertation and briefly describe two techniques to improve efficiency in model checking: mixed execution and ∆Execution. Finally, we list the contributions of this work and give an outline for the rest of this dissertation.

1.1 Challenges for software model checking

We start this section by introducing the key operations in program model checking. The challenge is to implement these operations efficiently.

The key operations in explicit-state model checking of programs are: (i) *execution of transitions*, (ii) *path exploration (backtracking)*, and (iii) *path pruning (state comparison)*. Execution of transitions refers to the operation that performs a deterministic step in the subject program, path exploration refers to the operation that performs exploration of *all* program paths created with non-deterministic choices, and path pruning refers to the operation that performs pruning of some of these paths based on some notion of equivalence, e.g., isomorphism of states that the model checker visits during the exploration. We later refer to path exploration just as backtracking (backtracking is part of path exploration) and to path pruning just as state comparison. It is important to point out that we focus on model checkers that perform backtracking by storing and restoring state and perform path pruning by comparing states during the state space exploration. (For a more detailed description of these operations, refer to Section 2.5.)

Model checking *programs* poses particular challenges. It is the implementation that defines the model, not the design. As a consequence, a deterministic step of the exploration can be potentially more expensive as it may require the execution of a complex fragment of sequential code. In addition, the states that the model defines are potentially larger, making backtracking and state comparison, which are two key model checking operations – whose performance depend on the size of the state – potentially slower.

1.2 Thesis statement

This dissertation proposes two techniques to improve time efficiency in explicit-state model checking of programs that dynamically allocate data. We call these techniques mixed execution and delta execution. Our thesis is that:

Mixed Execution and Delta Execution can be effective for reducing overall exploration time in the explicit-state space exploration of programs that manipulate dynamically allocated data.

1.3 Mixed Execution

Explicit-state model checkers of programs are typically designed to speed up operations for backtracking and state comparison. They often use a representation of state that contribute to making these key operations efficient. We call such representation *special state representation*. Making these operations efficient results in a more efficient exploration of *all* program paths. However, the use of special state representation makes *each* individual execution inefficient: the model checker needs to track every write in order to update the special state representation. This dissertation presents a technique to speed up the execution of transitions in model checkers that use special representation of state. The technique proposes the use of a different representation in certain parts of the execution.

We use the term *native state representation* to refer to a representation of state that favors execution, typically native to some environment. In the case of Java, for instance, native representation refers to the representation of state that the JVM uses.

The main idea of mixed execution is to execute *some* parts of the program natively, instead of on top of an environment that operates on special state representation. The technique can reduce overall exploration time by reducing time to execute transitions. With mixed execution, the model checker still compares, stores and restores the states as in its regular operation. The model checker executes natively only *deterministic blocks*, i.e., parts of the execution that have no thread interleavings or non-deterministic choices. It works by translating the state between different representations on the execution boundaries of these blocks.

4

```
 public class BST {
    private Node root;
    private int size;

 1:  public void add(int info) {
 2:    if (root == null)
 3:      root = new Node(info);
 4:    else
 5:      for (Node temp = root; true; )
 6:        if (temp.info < info) {
 7:          if (temp.right == null) {
 8:            temp.right = new Node(info);
 9:            break;
10:          } else temp = temp.right;
11:        } else if (temp.info > info) {
12:          if (temp.left == null) {
13:            temp.left = new Node(info);
14:            break;
15:          } else temp = temp.left;
16:        } else return; // no duplicates
17:    size++;
18:  }

    public boolean remove(int info) { ... }
 }

 class Node {
   Node left, right;
   int info; Node(int info) { this.info = info; }
 }
```

Figure 1.1: Binary Search Tree.

1.3.1 Mixed execution in JPF

We implemented mixed execution in Java PathFinder (JPF), a popular, general-purpose model-checker for Java programs. JPF implements its own JVM (that runs on top of a regular JVM) and uses special representation of state. Mixed execution translates the state from JPF to the state of an underlying JVM at the beginning of a deterministic block and performs the inverse translation at the end of that block. These two translations introduce an overhead, but the speedup obtained by executing on the host JVM can easily outweigh the slowdown due to the translations. JPF has slower execution not only because it uses special-state representation but also because it implements an interpreter on top of a regular virtual machine.

Figure 1.1 shows a binary search tree class that implements a set. Each BST object stores the

96: [2, 97]
97: [3, 98, −1]
98: [2, −1, −1]

mixed execution

add(1)

96: [3, 97]
97: [3, 98, −1]
98: [2, 99, −1]
99: [1, −1, −1]
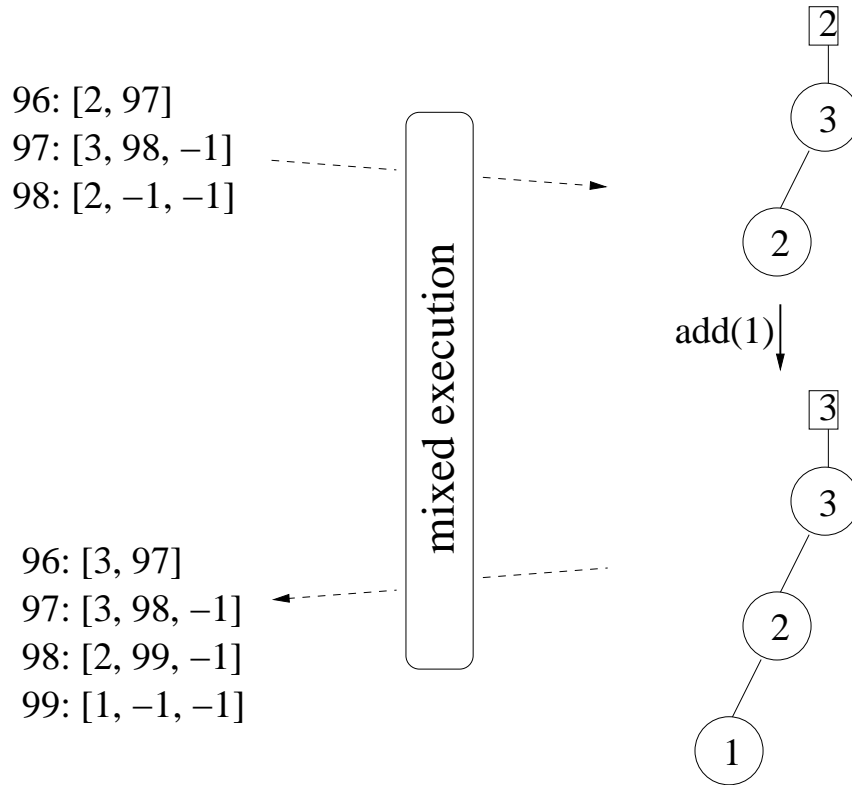
Figure 1.2: Running the method add(1) in JPF using mixed execution.

size of the tree and its root node, and each Node object stores an integer value and references to the two children. The BST class has methods to add and remove tree elements. A test sequence for the binary search tree class consists of a sequence of method calls, for example BST t = new BST(); t.add(3); t.add(2).

Figure 1.2 illustrates how mixed execution works in JPF when the model checker attempts to invoke the method add(1) against a binary search tree that the method sequence add(3); add(2) builds. Note from the figure that JPF represents heaps as array of integer arrays. Three integer arrays along with their references represent the pre-state of the tree. The references to these arrays, expressed with the integers 96, 97 and 98, are indexes on the array denoting the entire heap. The value -1 refers to the null reference. Field types distinguish between primitives and non-primitives.

Mixed execution intercepts the call add(1), translates the state to a native form shown in the upper-right corner and invokes the method natively. The resulting state, appearing in the bottom-right corner, is then translated back to the JPF state.

Although we evaluate mixed execution in the context of JPF, the main idea – executing parts

of model checking on different state representations – generalizes to other model checkers that use some special state representation, including AsmLT [Fou], BogorVM [RDH03], and SpecExplorer [VCST05].

1.4   Delta Execution

We next introduce ΔExecution, a technique that speeds up explicit-state model checking of software by applying symbolic model checking (SMC) on heaps. (See details on SMC in Section 2.3.)

SMC makes it possible to deal efficiently with sets of states to perform state space exploration but does not support efficiently operations on state with heaps. EMC makes it practical to manipulate concrete states including heaps but does not scale well in the presence of a large number of complex states. Delta Execution (ΔExecution) circumvents the inability that symbolic model checkers have to efficiently manipulate heaps symbolically. In essence, it provides a way for explicit-state model checkers to explore the state more efficiently with the use of sets.

The proposed technique leverages the fact that many paths a model checker traverses during a systematic exploration of the state space partially overlap. Conceptually, the technique enables sharing of the prefixes of computation among individual executions that have the same flow of control. Figure 1.1 shows a fragment of a class implementing a binary search tree that we use to illustrate the technique. Figure 1.3 shows individual instances of tree objects into boxes denoting set of states. The top of the figure shows a set including all the states that one can build with a binary search tree of size 3, containing values from 1 to 3. The bottom of the figure shows the state space reachable when adding the value 4 to each of these trees.

The key points to illustrate in this figure are as follows.

- Execution of the method call add(4) operates on sets of binary trees. ΔExecution defines a special form of execution that operates on sets of concrete states. It changes the dynamic semantics of the programming language to operate on sets of states. For example, we redefine the operator that compares the equality of two object references to compare the equality of two sets of object references.

- Execution may *split* the set of states on branching points. It is often the case that the

7
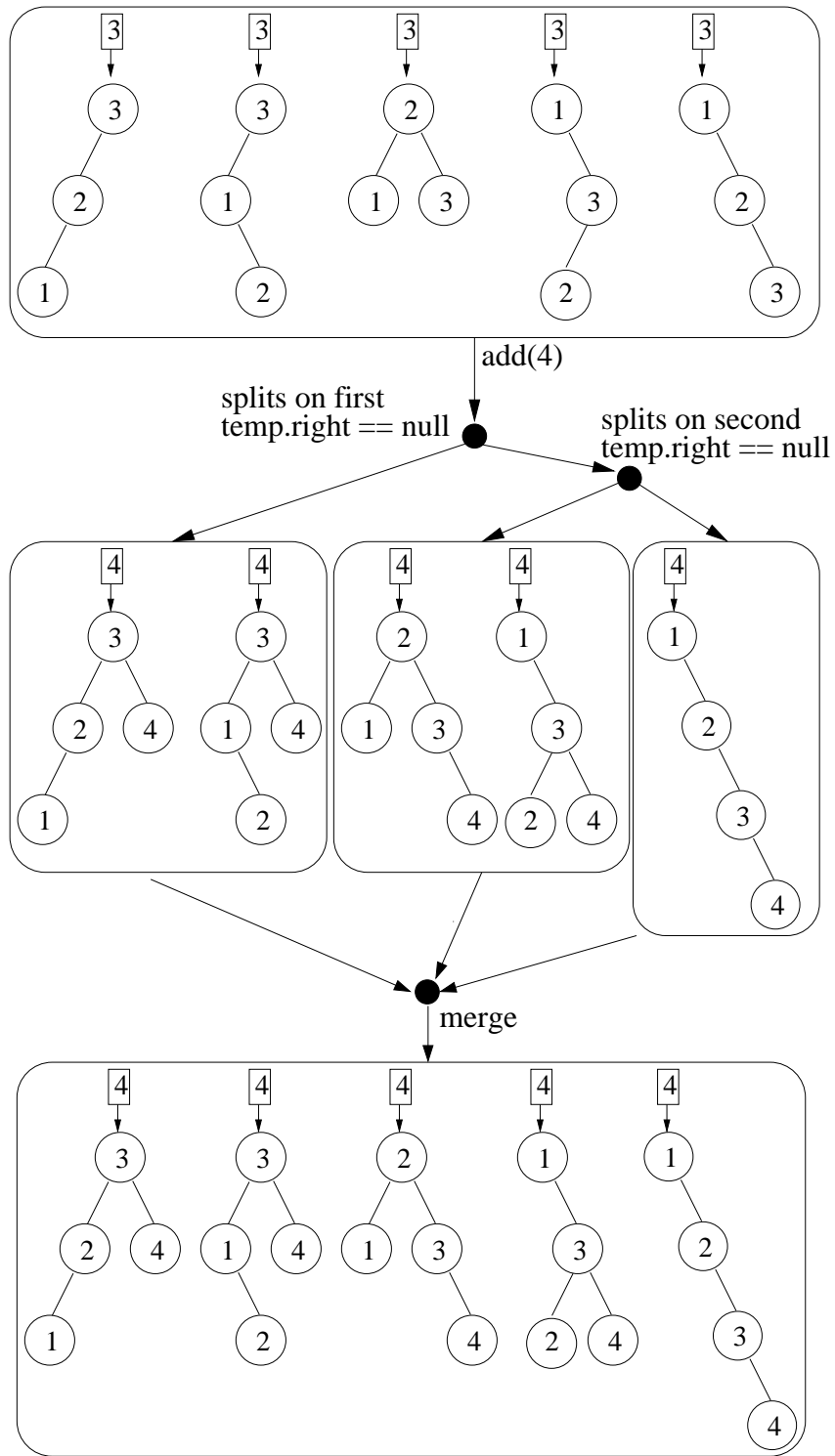
Figure 1.3: Executions of `add(4)` on a ΔState.

technique requires to "split" the set in order to enforce the invariant that all the individual states in the set follow the same execution path. This happens in branching points whenever

it is not possible for all the states in the set to evaluate the branching condition to the same value. The top-two dark bullets correspond to splits (branching points) during the execution of the body of the method `add`. Effectively, split points originate non-deterministic choices in the exploration. We informally say that a "path splits" to denote that the set splits and a non-deterministic choice has been created to explore each path in isolation.

- Multiple sets of states *merge* to construct a post state. Merging allows to construct new sets from existing ones. Determining when to merge states is an important decision for the implementation of the technique. We chose to merge on the boundaries of method invocations as the figure suggests.

The key aspects regarding the performance of ΔExecution are as follows.

- ΔExecution may reduce time to *execute transitions*. This happens because it is sometime possible to use constants valid across all states in the set. For example, an operation that reads the field `size` in either the pre or post state takes constant time, instead of time linear in the number of states.

- ΔExecution can reduce time to *compare states*. Instead of comparing states one-by-one, the use of sets for state-space exploration allows the model checker to compare sets of states. The overlapping that exist between the states in the set enables a more efficient comparison.

- ΔExecution can reduce the overall time for *backtracking*. The reduction in the number of executions enables less backtracking. Figure 1.3 illustrates that ΔExecution traverses only 3 of the 5 paths that lead each individual pre to a post-state. The paths only split when it is not possible for all the states in the current set to evaluate the branching condition to the same value.

1.5   Summary of evaluation

This section presents a scenario of exploration that we use to evaluate the techniques and some of our experimental results.

```
// N bounds sequence length and parameter values
public static void main(int N) {
  BST bst = new BST(); // empty tree
  for (int i = 0; i < N; i++) {
    int methNum = Verify.getInt(0, 1);
    int value = Verify.getInt(1, N);
    switch (methNum) {
      case 0: bst.add(value); break;
      case 1: bst.remove(value); break;
    }
    stopIfVisited(bst);
  }
}
```

Figure 1.4: Driver used to explore exhaustively BST objects within given bounds.

1.5.1 Example of bounded-exhaustive exploration

We now present an example that illustrates how to use an explicit-state model checker to perform *bounded-exhaustive exploration*, a common scenario of use of model checkers that we apply in the evaluation of our techniques.

In this scenario, the model checker exhaustively explores all sequences of method calls in the subject under test up to some bound [XMSN05,VPP06,DB06]. Such exploration does not actually enumerate all sequences but instead uses state comparison to prune sequences that produce the same states.

Figure 1.4 shows an example driver program that enables a model checker to systematically explore different states of the tree using bounded-exhaustive exploration. The driver creates the initial state of the binary search tree and exhaustively explores sequences (up to length N) of the methods add and remove (with values between 1 and N). The driver non-deterministically selects different methods and input values using the library method getInt(int lo, int hi), as shown in Section 2.1.

This driver discards from further exploration any sequence that results in a state that has already been visited: it uses the library method stopIfVisited(Object root) that ignores the current execution path and forces *backtracking* (to a preceding choice point) if the state reachable from root has already been visited in the exploration. Note that the *comparison of states* is performed only at the method boundaries (not during method execution), which naturally partitions an exe-

cution path into subpaths that each cover execution of one method invocation. As in other related studies [XMSN05, VPP06, dPX$^+$06], this driver explores states in breadth-first order. A depth-first exploration could miss parts of the state space since state comparison could eliminate a state with a shorter sequence in favor of a state with a longer sequence.

Figure 1.5 shows a graph that represents the state space the model checker produces when exploring the states of BST with this driver and $N = 3$. Nodes denote BST states, and edges method invocations on BST objects. To simplify the illustration, we omit from the graph edges that correspond to invocations of delete as each "delete" edge is either a self-loop or leads to some existing state.

Each path in this graph denotes a method sequence consisting of invocations of add against the BST object that the model checker creates using the driver. For instance, it first constructs an empty tree and then invokes non-deterministically the method add with values 1, 2, and 3. The states produced with these calls appear in the graph at depth 1. The presence of states with more than one incoming edge indicates that some method sequences lead to a previously visited state, i.e., the state comparison the operation stopIfVisited performs is in effect.

In mixed execution, we select some of the methods reachable from those the driver invokes to execute in native mode. Also, the driver in ΔExecution merges sets of states after the completion of every iteration. Recall that we use breadth-first exploration as search strategy.

1.5.2 Summary of experimental results

We first evaluate mixed execution on seven subject programs. The experimental results show that mixed execution can improve the overall time for state exploration in JPF from 1.01x to 1.73x (with median 1.13x), while improving the time for execution of deterministic blocks from 0.91x to 3.05x (with median 1.64x). We optimized the baseline translation of state by translating only the parts of the states which are actually reachable from the execution. We refer to the baseline translation as eager, and refer to its optimized version as lazy. Our experiments show that lazy translation can improve the eager mixed execution from 1.03x to 1.35x (with median 1.08x).

We also evaluate mixed execution on a larger case study containing bugs. The exploration time on this experiment reduces from 1.14x to 1.41x (with median 1.23x). It is important to point
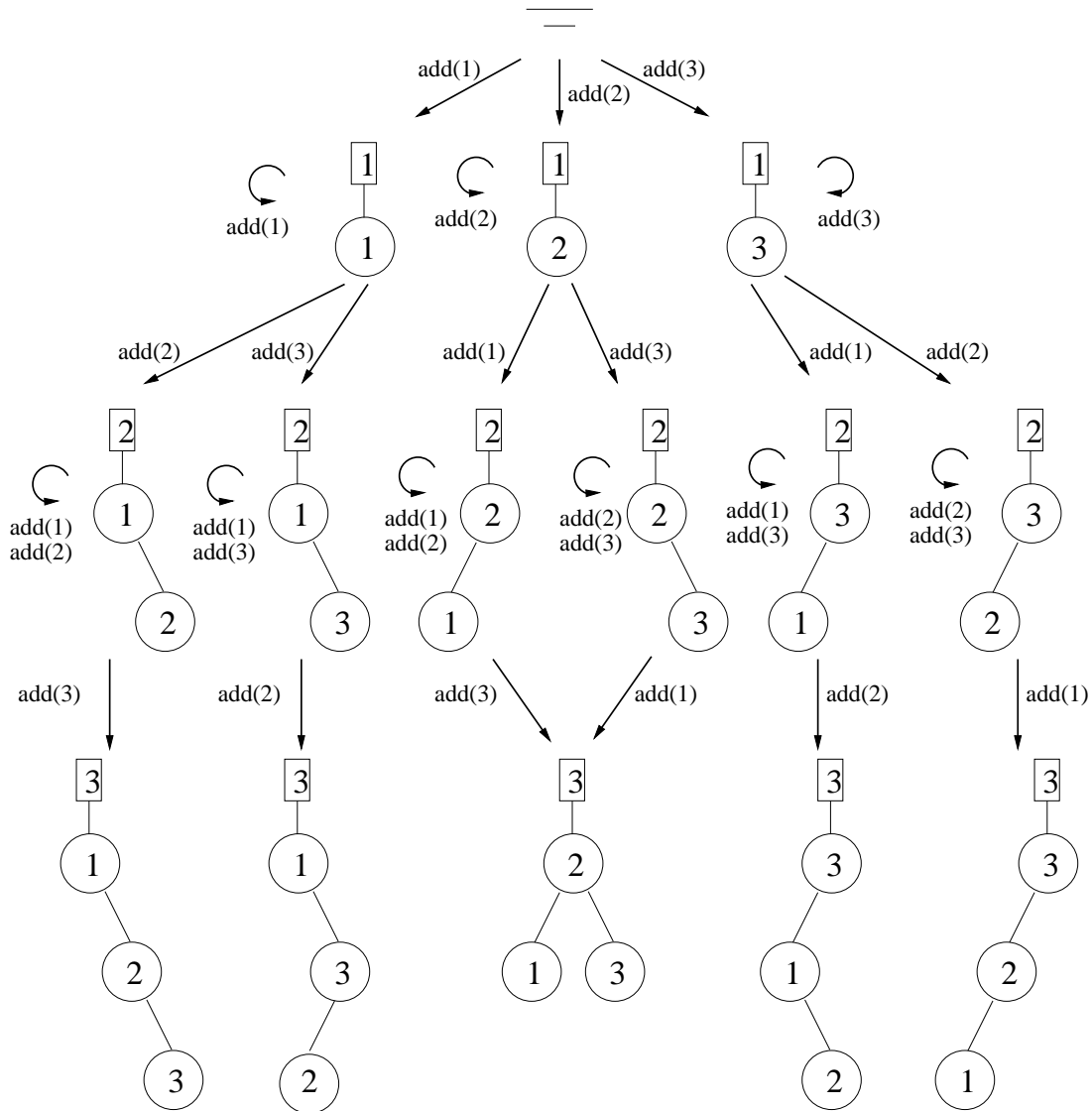
Figure 1.5: Bounded state space of binary search tree. We omit edges that correspond to invocations of `delete`: each "delete" edge is either a self-loop or lead to some existing state.

out that mixed execution reduces the overall exploration time by reducing the execution time for deterministic blocks. Mixed execution does not affect the order of exploration, the number of explored states, or any other aspect of the state exploration. We implemented mixed execution in the JPF model checker.

We implemented Delta Execution in two model checkers: JPF and BOX (*Bounded Object eXploration*). JPF is a popular general-purpose model checker for Java programs, while BOX is a specialized model checker that we developed for efficient exploration of sequential Java programs.

	Execution	Path exploration	Path pruning
Mixed execution	✓	-	-
ΔExecution	✓	✓	✓

Table 1.1: Addressing time efficiency in explicit-state model checkers.

The goal is to evaluate the effectiveness of the technique in model checkers designed with different principles. The results show that ΔExecution improves the overall exploration in both tools, but the improvements are due to different factors. The improvement in JPF is mainly due to execution and backtracking, while in BOX it is mainly due to the optimized comparison of state.

We first evaluate ΔExecution on ten simple subject programs. The results show that ΔExecution improves the exploration time from 0.88x to 126.80x (with median 5.60x) in JPF and from 0.58x to 4.16x (with median 2.23x) in BOX, while taking from 0.46x to 11.50x (with median 1.48x) less memory in JPF and from 0.18x to 2.71x (with median 1.18x) memory in BOX. We also evaluate ΔExecution for one case study of large size including errors, where the exploration time improved from 0.88x to 2.04x (with median 1.72x).

Visser et al. [VPP06] recently proposed and implemented in JPF several non-exhaustive explorations. Their results on four subject programs showed that *abstract matching* achieved the best structural code coverage. ΔExecution improves exploration time for abstract matching from 0.92x to 6.28x (with median 4.52x) in JPF. The relative decrease of improvement in ΔExecution – from 5.60x in exhaustive to 4.52x (median values) in non-exhaustive – is due to the sharp reduction in the number of states and executions when using abstract matching.

1.6 Contributions

Our contributions are as follows.

- Ideas of mixed execution and ΔExecution. We proposed two techniques to improve time efficiency of the key operations of explicit-state model checking. Table 1.1 relates each technique with the operation it improves. Mixed and ΔExecution can improve time for the execution of transitions, while ΔExecution can, in addition, improve time for the path exploration (backtracking) and path pruning (state comparison).

13

- Proposal of a data structure that represents efficiently sets of single-rooted heaps. The proposed data structure contains operations that allow efficient merging and splitting of sets of concrete heaps (see Section 1.4).

- Implementation of mixed execution in JPF, and implementation of ΔExecution in both JPF and BOX. We modified the JPF model checker to implement the two techniques and implemented BOX, a custom built model checker specialized in sequential code.

- Evaluation on several subjects and a larger case study. We evaluated the techniques using data structures from a variety of sources [SM03, CS04, Qad, PE05, XMSN05, DB06, VPP06] and one larger case study, the AODV routing protocol [PR99].

Chapter 2

Background

This chapter presents concepts and terminology to support the discussion of the remaining chapters. We first illustrate basic concepts in model checking with a simple example. We then discuss two important orthogonal aspects in the design of a model checker: (i) how the model checker represents state and (ii) what kind of input language it uses. These aspects will be important in the presentation of our techniques. Then, we highlight the key operations involved in a class of model checkers that operate on actual code, and discuss the design and implementation of these operations in a specific model checker, Java PathFinder (JPF) [LV01,VHB$^+$03]. Finally, we discuss the AODV case study that we use to evaluate the techniques in this dissertation.

2.1 Model representation

A *model* is a description of a system that can serve to predict or explain its behavior [CGP99]. A model describes a set of *states* and a *transition relation* which tells how the system transits from one state to another. Once one builds a model, it is possible to argue whether or not a *property*, characterizing some expected or unexpected behavior of the system, holds for that model. *Model checking* is the process of exploring the state space of a model to find property violations or to demonstrate that none exist. *Model checkers* are tools that automate this process.

Non-determinism is central to the construction of models. It provides a framework for defining the semantics of language constructs that does not depend solely on the input state [Gun92,Win93]. Non-determinism can originate from the thread interleavings of a concurrent program run. The interleavings depend on an external (and uncontrolled) environment; it is thus not possible to determine the interleaving prior to an execution. Such environment includes, for example, the CPU load. Non-deterministism is an abstraction that one uses to model the possible interleavings

of a concurrent program. Non-determinism can also originate from the programmer as a means to model the choice of inputs the program can take. In summary, non-determinism enables the expression of different choices of control and input data that the program can exercise. This dissertation focuses on sequential programs and uses non-determinism only to express the choice of inputs.

We use the following non-deterministic program to build a model for showing different aspects of software model checking. This program tests a one-bit adder, provided by the operation +.

```
1:  x = Verify.getInt(0, 1);

2:  y = Verify.getInt(0, 1);

3:  (result, overflow) = x + y;

4:
```

Variables x, y, result, and overflow are 1-bit integers that range over the domain $D_1 = 0..1$ and are initialized to 0, while the implicit program variable pc ranges over $D_2 = 1..4$ and is initialized to 1. Variables x and y model the input to the adder, variables result and overflow model the outcome of the addition, and variable pc models the program counter (simply the line number here). The operation Verify.getInt(int lo, int hi) introduces a non-deterministic choice point to return a value between lo and hi as input to x and y. In other words, instead of always producing a single execution like deterministic programs, this non-deterministic program can produce 4 executions corresponding to the 4 different combinations of values assigned to x and y.

We need to take into account the fact that the domains of variables in this program are finite. For example, the addition of 1 and 1 results in a value outside the domain D_1. To deal with overflows of this kind, we make arithmetic operations to return pairs. The first element denotes the result of the operation when no overflow arises in the calculation, the second element is a flag indicating whether or not the arithmetic operation resulted in overflow.

The triple $\langle S, S_0, R \rangle$ defines a model. The set S denotes the states in the model, $S_0 \subseteq S$ denotes the initial set of states, and $R \subseteq S \times S$ denotes the transition relation. The state of the model includes the values for the input variables x and y, the output variables result and overflow, and the program counter pc. The state consists of assignments to these variables. We use a sequence of single-digit integers to represent a state, e.g., 10002 refers to a state having x equals to 1, having

16

y, `result`, and `overflow` equal to 0, and `pc` equals to 2. Note that this state is in fact reachable from the initial state 00001. One possible model for this program follows.

$$S \equiv D_1^4 \times D_2$$

$$S_0 \equiv x = 0 \land y = 0 \land result = 0 \land overflow = 0 \land pc = 1$$

$$R \equiv (pc = 1 \Rightarrow (x' = 0 \lor x' = 1) \land y' = 0 \land result' = 0 \land overflow' = 0 \land pc' = 2) \land$$
$$(pc = 2 \Rightarrow (y' = 0 \lor y' = 1) \land x' = x \land result' = 0 \land overflow' = 0 \land pc' = 3) \land$$
$$(pc = 3 \Rightarrow x' = x \land y' = y \land result' = x \oplus y \land overflow' = (x \land y) \land pc' = 4)$$

The cartesian product of the domains of all program variables forms the set S from the model. The initial set of states is the singleton set, corresponding to the initial assignment of each program variable. Typically, the model encodes the transition relation implicitly, i.e., in some form from which it is possible to determine the potential successor(s) for each given state or to build state pairs, which correspond to pre- and post-states of each program transition. The model above encodes the transition relation as a successor function. This function takes as inputs an assignment of program variables, including the program counter and produces another assignment that corresponds to the post-state. Note that we use a boolean function for this encoding and that we express non-determinism with disjunction. The symbol $\Rightarrow$ denotes logical implication and the symbol $\oplus$ refers to the exclusive-or boolean operator. The program addition results in overflow when both x and y store the value 1.

Now, it is possible to express properties about the model. For instance, the user of a model checker may want to check that execution cannot reach the overflow state from the initial state. Note that this property does not hold for this model since there is a path in the model from the initial state to a state $s \in \{ abc1d \mid a, b, c \in D_1 \land d \in D_2 \}$.

2.2   Explicit-State Model Checking (EMC)

Explicit-state model checkers represent states and the transitions between them *explicitly*, i.e., they enumerate every possible state pair of the transition relation. Below, we show one explicit encoding of the transition relation R.

$$\{(00001, 00002), (00001, 10002),$$

$(00002, 00003), (00002, 01003), (10002, 10003), (10002, 11003)$

★ $(00003, 00004), (01003, 01104), (10003, 10104), (11003, 11014)\}$

The line with the label ★ illustrates state transitions that relate to line 3 of the program. Figure 2.1 shows a graph that explicitly represents the previous model. Nodes in this graph denote states while edges denote the transitions between them. We label the edges with the statement the program executes to carry out a state transition. Note that we omit from the label the implicit assignment to the variable `pc`. Explicit-state model checking works by exploring the nodes of this graph in some specific order to find property violations. (The graph is typically not available prior to the exploration, but constructed on-the-fly as the model checker finds new states.)
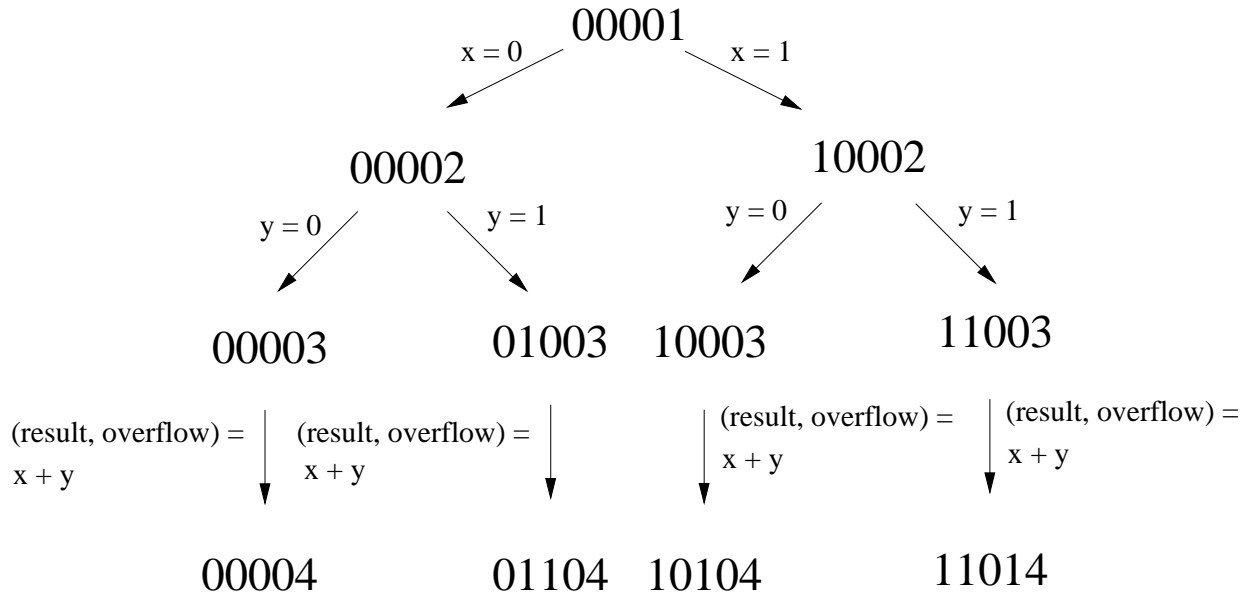
Figure 2.1: Explicit state representation of the model.

It is important to highlight that a model checker can assist with the construction of tests. The sequence of experiments a test includes corresponds to the decisions made in the state space exploration. The oracle of the test corresponds to the properties that the model checker verifies. In this case, the property that the variable `overflow` should have value 0 in any reachable state. For instance, the test `x = 1; y = 1; (result, overflow) = x + y; assert overflow == 0;` corresponds to one path in the graph of Figure 2.1 and reconstructs the state 11014 that exposes an overflow. In principle, the model checker can check the property after the execution of every transition. Here, for simplicity, the test includes only the check for the last transition. (The test implements the

check with a state assertion.) A state invalidates the assertion above if the predicate `overflow ==`
`1` holds in that state. An assertion violation indicates that the test has failed.

2.3   Symbolic Model Checking (SMC)

Symbolic model checking [JEK+90, CGP99, McM92] presents an approach to deal with the huge
state-spaces that model checking can produce. SMC enabled a breakthrough in software verification
as it provided a significantly more efficient exploration than explicit-state model checking. More
precisely, SMC represents state symbolically, encoding *sets of concrete states*. The target of SMC
was originally hardware verification. Even nowadays, it is mostly used for hardware or abstractions
of software, as it requires specific state representations.

   Symbolic model checking represents set of states and the transition relation as boolean functions.
Such encoding enables the construction of special functions that map set of states to another.
Symbolic model checking proceeds by finding fixpoints of such functions, where fixpoint is an input
of a function for which the function produces the same value as result, i.e., x is a fixpoint of f if
$x = f(x)$. Intuitively, we say that such special functions "accumulate information": an application
of the function approximates the result to the solution, i.e., to a fixpoint. Section 2.3.1 shows the
conditions for the existence of fixpoints and also illustrates symbolic model checking on our 1-bit
adder example.

   Conceptually, one can define a function that calculates a superset of the initial set of states, with
the property that all elements in the initial and final sets must satisfy some property. The transition
relation indicates which additional states to include in the final set (e.g., parent, children).

   Typical implementations of SMC encode boolean functions with Ordered Binary Decision Dia-
grams (BDDs for short) [Bry92]. BDDs provide a canonical and compact representation for boolean
functions, and enable efficient operations over them. These operations are central to making sym-
bolic model checking viable.

2.3.1   SMC in more detail

This section illustrates how symbolic model checking works on the 1-bit adder example program.
It shows how to encode sets and relations as boolean functions, how to express properties in terms

of such functions, and how to calculate their fixpoints.

Symbolic Model Checking (SMC) represents states and transitions as boolean functions. For example, the boolean function $\hat{S}_0(\vec{x}) \equiv \neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge \neg x_5 \wedge \neg x_6$ represents the initial set of states for the model appearing in Section 2.1. This function uses boolean variables to encode program variables: x_1 encodes `x`, x_2 encodes `y`, x_3 encodes `result`, x_4 encodes `overflow`, and x_5 and x_6 together encode `pc`. The symbol $\vec{x}$ is a shorthand for the sequence of boolean variables $x_1, ..., x_6$, and the symbol ˆ indicates that a definition uses a boolean encoding.

Note that with the exception of `pc`, the program variables only take values 0 and 1 in the original program. An assignment of *false* to x_1, for example, corresponds to an assignment of 0 to the program variable `x` (similarly, *true* corresponds to 1). Boolean vectors can encode variables from other domains too: one can use the bijection $\{0, 1\}^m \to D$ to encode the elements of a domain D with 2^m elements (with $m > 0$). As a matter of fact, the `pc` variable can store any value from 1 to 4 in the example program. It requires two boolean variables to encode the four possible assignments; we use the bijection $\{00 \mapsto 1, 01 \mapsto 2, 10 \mapsto 3, 11 \mapsto 4\}$ as the encoding for `pc`.

A boolean formula over these variables thus encodes a set of concrete states. In this example the function $\hat{S}_0$ denotes a set including a single concrete state, i.e., a singleton set.

The boolean function $\hat{R}(\vec{x}, \vec{x'}) \equiv t_1(\vec{x}, \vec{x'}) \wedge t_2(\vec{x}, \vec{x'}) \wedge t_3(\vec{x}, \vec{x'})$ encodes the transition relation R. The definition of $\hat{R}$ uses two sequences of variables, one relative to the the current state, $\vec{x}$, and the other relative to the next state, $\vec{x'}$. Below we show the definitions of the functions t_1, t_2, and t_3, which conceptually encode each individual transition of the illustrative program.

$t_1(\vec{x}, \vec{x'}) \equiv (\neg x_5 \wedge \neg x_6) \wedge (\neg x_5' \wedge x_6') \wedge n(2) \wedge n(3) \wedge n(4)$

$t_2(\vec{x}, \vec{x'}) \equiv (\neg x_5 \wedge x_6) \wedge (x_5' \wedge \neg x_6') \wedge n(1) \wedge n(3) \wedge n(4)$

$t_3(\vec{x}, \vec{x'}) \equiv (x_5' \wedge \neg x_6') \wedge (x_5' \wedge x_6') \wedge (x_3' = x_1 \oplus x_2) \wedge (x_4' = x_1 \wedge x_2) \wedge n(1) \wedge n(2)$

We use the boolean function $n(i) \equiv x_i' = x_i$ to express that the variable x_i did not change value in the transition. Note that transition t_1 does not use variable x_1', and transition t_2 does not use variable x_2'. This indicates that the function does not depend on that variable: x_1' can be either true or false after taking t_1. Also note that the transition t_3 identifies an overflow state when both x_1 and x_2 are true.

The encoding $\hat{R}$ enables the creation of functions on set of states, $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$, that

20

conceptually express how a set of states evolve when making program transitions. Symbolic model checking proceeds by finding fixpoints of such functions. A fixpoint of τ is an input for which τ returns the same value, i.e., some set of states p for which $\tau(p) = p$. Assuming an application of τ results in a superset of the input, one can, intuitively, calculate a fixpoint with iterative applications of τ starting from a a minimum element from its domain $\mathcal{P}(S)$. We call such fixpoint the least fixpoint since it is the fixpoint which is less or equal to any other fixpoint in the domain; in this case, the smallest set in $\mathcal{P}(S)$ satisfying the constraints τ expresses. The sequence:

$$false \subseteq \tau(false) \subseteq ... \subseteq \tau^i(false) = \tau^{i+1}(false),$$

illustrates the iterative calculation of the fixpoint $\tau^i(false)$, where the exponent in τ means number of iterations. In addition to assuming that τ calculates a superset of its input, note that we also assume that there is finite number of elements in $\mathcal{P}(S)$. These conditions are sufficient to guarantee that the iterative procedure generates a fixpoint [Tar55].

Now that we understand better how to compute fixpoint of functions, such as τ, let us show with a concrete example how to apply the concept to model checking. First, we need to construct a recursive function on set of states that characterizes how information flows. For example, we use the function τ below [McM93] to accumulate the set of reachable states in the state space exploration. Assume that the context of call to τ binds the variables in $\vec{x}$.

$$\tau_0 \; \equiv \; \lambda Z.\lambda\vec{x'}. \; (\hat{S}_0 \; \vee \; (\exists \; \vec{x}. \; Z \; \wedge \; \hat{R}(\vec{x}, \vec{x'})))$$

$$\tau \; \equiv \; \lambda Z. \; (\tau_0 \; Z \; \vec{x})$$

This function satisfies the conditions described above, i.e., it produces a superset of the input states and it also operates on a finite domain. Therefore, one can iteratively apply this function to find the set of all reachable states from the model. More precisely, the set of all reachable states from the initial set of states $\hat{S}_0$. Note that the variables in $\vec{x}$ which are free in $\hat{R}$ bind tighter in the scope of the existential quantifier, i.e., the lambda application does not substitute such variables.

We show next the two initial iterations of the fixpoint computation. We make use of the notation $f_{|x_i \leftarrow v}$ to denote the substitution of x_i with v in f, and the property that $\exists x.f \equiv f_{|x_i \leftarrow 0} \vee f_{|x_i \leftarrow 1}$.

$$\tau(false) = \hat{S}_0 \; \vee \; (\exists \; \vec{x}. \; false \; \wedge \; \hat{R}(\vec{x}, \vec{x'})) = \hat{S}_0$$

$$\tau^2(false) = \tau(\tau(false)) = \tau(\hat{S}_0)$$

$$= \lambda\vec{x'}. \; (\hat{S}_0 \; \vee \; (\exists \; \vec{x}. \; \hat{S}_0 \; \wedge \; \hat{R}(\vec{x}, \vec{x'})))(\vec{x})$$

$$= \hat{S}_0 \ \vee \ \lambda\vec{x'}.(\exists \ \vec{x}. \ \hat{S}_0 \ \wedge \ \hat{R}(\vec{x}, \vec{x'}))(\vec{x})$$

$$= \hat{S}_0 \ \vee \ \lambda\vec{x'}.(\exists \ x_1..x_5.((\hat{S}_0 \ \wedge \ \hat{R}(\vec{x}, \vec{x'}))_{|x_6 \leftarrow 0} \wedge (\hat{S}_0 \ \wedge \ \hat{R}(\vec{x}, \vec{x'}))_{|x_6 \leftarrow 1}))(\vec{x})$$

$$= ...$$

$$= \hat{S}_0 \ \vee \ \lambda\vec{x'}.(\neg x'_2 \wedge \neg x'_3 \wedge \neg x'_4 \wedge \neg x'_5 \wedge x'_6)(\vec{x})$$

$$= \hat{S}_0 \ \vee (\neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge \neg x_5 \wedge x_6)$$

Note that the set of states $\tau^2(\textit{false})$ given by the final equality includes the initial state and the two states reachable after the first program transition.

Binary Decision Diagrams

This section discusses the use of Binary Decision Diagrams (BDDs) to encode boolean functions.

A binary decision diagram is as a direct acyclic graph (DAG) with a single root node. It contains two terminal nodes: 0-terminal and 1-terminal. A boolean variable labels each non-terminal node, and each node has exactly two child nodes called low and high child; 0 labels the low edge and 1 labels the high edge. The low (respectively, high) edge denotes an assignment of the value 0 (respectively, 1) to the variable that labels the source node. Any path from the root node to the 1-terminal (respectively, 0-terminal) node corresponds to an assignment of variables for which the boolean formula evaluates to true (respectively, false). Variables that do not occur in such a path are not relevant in the result; they can take any value.

We say that a BDD is *ordered* if variables appear in the same order across any path from the root node [Ake78,Bry92,CGP99]. We say that a BDD is *reduced* if it merges any isomorphic graph (i.e., redirects any incoming edge of one isomorphic graph to the other) and eliminates any node whose low and high edges lead to the same node (i.e., redirects any incoming edge to such a node to the target node of its low and and high edges). An ordered and reduced binary decision diagram provides a compact and canonical representation for a boolean function [Bry92]. The fact that the representation is unique has certain advantages. For example, checking whether a boolean function is satisfiable boils down to comparing a BDD of that function with one for *false*.

Figure 2.2 illustrates an example BDD that encodes the transition function $t_1(\vec{x}, \vec{x'})$. Note that the BDD is reduced and ordered; it uses the variable ordering $x_5, x_6, x'_5, x'_6, x_1, x'_1, x_2, x'_2, x_3, x'_3, x_4, x'_4$. The selection of variable ordering can lead to significant reduction of size of BDDs. However, finding
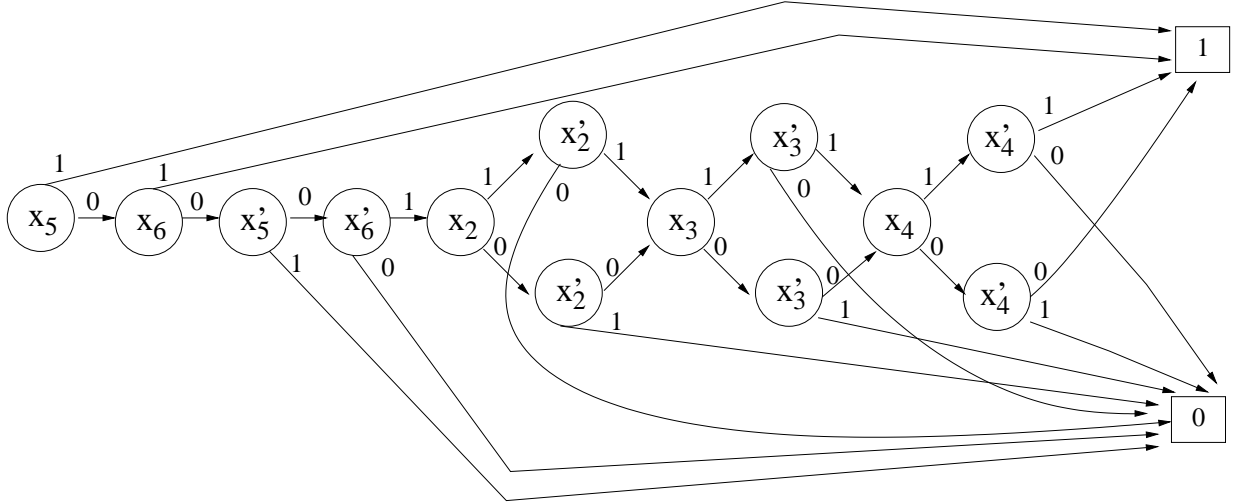
Figure 2.2: Binary Decision Diagram for boolean function $t_1(\vec{x}, \vec{x'})$ using variable ordering $x_5, x_6, x_5', x_6', x_1, x_1', x_2, x_2', x_3, x_3', x_4, x_4'$. The node with the x_5 label is the root, and the square nodes with labels 0 and 1 are terminals.

a good ordering is a problem of high complexity [BW96].

BDDs can implement boolean operations efficiently. For example, it is possible to construct BDDs for t_2 and t_3 and compute efficiently $\hat{R}$ by taking the conjunction of the predicates that define the transition relations t_1, t_2, and t_3. BDDs also support other important operations to model checking. The implementation of the substitution operator $f_{|x_i \leftarrow v}$, for example, simply removes any node with an x_i label and redirects their incoming edges to the low node (i.e., the node that the low edge leads to) in case $v = 0$, or to the high node otherwise.

Symbolic model checking has been mainly used to verify hardware and abstraction of programs. In these cases, models typically have fixed state. (See Section 2.3.) Model checking proceeds by finding fixpoint of functions in a similar fashion as we show in Section 2.3.1 for calculating the set of reachable states. Note that the program that we show in Section 2.1 does not allocate memory; the states associated to executions of this program have fixed size.

2.3.2 Limitations of SMC with BDDs

SMC has been successfully applied in hardware verification [CGH+95, JEK+90, McM00]. Like software, hardware circuits can model state, e.g., sequential circuits have memory. Unlike software, a hardware circuit models only a fixed amount of state. A software analogy to a hardware circuit is

a program that does not allocate data; it contains only a fixed number of primitive-type variables and pre-allocated arrays. In essence, the hardware state has fixed size and is flat; it does not model the heap.

In practice, software often makes use of memory allocation (e.g., with the creation of objects in object-oriented programs). This observation makes an important practical difference between hardware and software verification in SMC. The hardware state enables simple encodings of the state and the transition relation with boolean functions. Unlike hardware verification, software verification does require to encode *heap operations* such as reading or writing to an object field or the allocation of a new object. These operations do not translate into efficient BDD operations. This limitation prevents the direct use of SMC for software that manipulates dynamically allocated data, such as object-oriented programs.

It is important to note that BDDs can represent states with heaps. For example, BDDs have been used to support efficient implementations of program analysis [WL04, LH04]. In that context, BDDs represent analysis information as sets and relations. More precisely, BDDs represent over-approximations of the set of reachable heaps. However, such applications do not use BDDs to represent states for *concrete execution*. Section 2.5 shows that execution is an important step in model checking of code. Concrete execution requires to index into the state to read and write data; it also requires dynamic object allocation and garbage collection. In addition, the model checker needs to read the heap to compare states based on heap symmetry [BKM02, CGP99, Ios01, LV01, MD05]. To the best of our knowledge we do not know of any work that has used BDDs as a representation of state for regular execution (as opposed to abstract interpretations of code). It remains to investigate how BDDs or other similar representation can perform with our techniques.

2.4   Notes about the model checker input language

We now discuss the role of the input language in the design and implementation of model checkers.

For a long time, software model checkers have been applied to verify models written in languages suited for specification and design. Examples of such model checkers and their languages include FDR with CSP [Hoa78, Ros94], NuSMV and its input language [CCGR99], and SPIN with Promela [Hol97]. These model checkers take as input models that express important design deci-

sions of an algorithm that are amenable to model checking, and leave out implementation details that, in principle, should not influence verification. Model checking designs is very important: the later one finds an error in the design the more expensive it is to fix the error.

Following a different approach, Godefroid [God97] proposed in 1997 Verisoft, a model checker that operates directly on C programs. The motivation is that finding errors directly on implementation is also very important. Since Verisoft, many other model checkers have been developed to operate on actual code. Examples include Bandera [CDH+00], BogorVM [RDH03], CMC [MPC+02], JCAT [DIS99], JNuke [ASB+04], JPF [LV01, VHB+03], SpecExplorer [VCST05], and Zing [AQR+04]. These tools have shown practical use in software verification. CMC, for example, a model checker for the C language, was used to verify Linux implementations of networking code and file systems [MPC+02, ME04, YTEM04]. As another example, JPF was used to verify a real time scheduling kernel [PVE+00] and several networking protocols [dSM06, dLM07, AG06].

It is important to point out that the target application influence the language design and ultimately model checking. For example, CSP processes do not have state as it models only the concurrency aspect of software (but CSP-Z [MS01] proposes an extension to support state). In addition, neither NuSMV nor Promela support object allocation (but Promela supports dynamic process creation). In the case of an implementation model the states correspond directly to program states, including stack, heap, static area, and the program counter; and the transitions of the model correspond to changes in program state, e.g., an assignment to a variable in the stack or a branch decision taken in the control-flow (as it modifies the program counter). Model checking actual code proceeds by exploring the different paths of the program, which requires *executing actual code.*

2.5   Key operations in explicit-state model checking

We already showed that the model checker can either represent state explicitly or symbolically. Also, we discussed that model checkers can take models with different level of details, e.g. design or implementation models. This dissertation focuses on explicit-state model checkers that take implementation models as input. This section describes, in more detail, the key operations for this kind of model checkers as appeared in Section 1.1.

- **Execution**. This operation defines how the model checker carries out a deterministic step, i.e., how the model checker runs a sequence of transitions that have no non-deterministic choice. For example, code can run on top of an interpreter or directly on the target machine.

- **Path exploration (backtracking)**. The model checker needs to remember which non-deterministic choices it made during the execution of one path and also to provide a mechanism to systematically explore the remaining choices, which correspond to uncovered paths. To explore an uncovered path the model checker needs to restore the state to the point where it made the choice and continue the exploration from that point. We call the restoration step *backtracking*. For simplicity, we will refer to the entire operation just as *backtracking* from hereon.

- **Path pruning (state comparison)**. This operation enables a model checker to prune paths from the exploration based on some specific criteria. Several criteria have been proposed in the past [CKL04, God96, DB06, VPP06]. We will focus our attention to the one based on the isomorphism of states [VHBP00, BKM02, RDHI03] as this is the predominant technique for pruning paths during the state space exploration of sequential programs. In this approach, the model checker prunes the path when it observes that the execution of that path produces a state that has been visited in the exploration of previous paths. For this reason, we will refer to path pruning just as *state comparison* from hereon.

Next, we discuss some implementation aspects of the operations above.

Execution

Some software model checkers use native representation of state in order to speed up execution. With native states, the model checker executes code using the same environment as a regular execution. In the case of C, the model checker runs the compiled code for the program on top of the operating system. In the case of Java, the model checker runs the code on top of a regular JVM. Examples of model checkers that follow this design include Verisoft [God97], CMC [MPC$^+$02, ME04, YTEM04], and BOX (see Section 4.3.2). Verisoft was the first model checker to directly analyze the implementation code, specifically code written in the C language; CMC has been used

to model check Linux implementations of networking code (e.g., AODV and TCP) and file systems; and BOX is a model checker for sequential Java code used to evaluate ΔExecution.

It is important to note that this approach does not offer to the model checker a unified view of the memory. For example, CMC does not keep track of the locations of the heap objects that a program execution allocates. It reads state by following roots of heap objects which are available to a test driver. A unified view of memory can be important to the efficient implementation of memory-intensive operations, such as state comparison or backtracking. It enables the model checker to manipulate state directly. We use the term *special state representation* in reference to the representation of state that a model checker uses so that it can access state directly. JPF, for example, represents the heap as a single array of integer arrays. It can efficiently restore parts of this array for backtracking, and efficiently compare two instances of such arrays (or fragments of the arrays) for state comparison.

In summary, native state representation can be helpful to execute transitions, special state representation can be helpful for defining memory intensive operations such as state comparison and backtracking.

Path exploration (backtracking)

We now discuss how EMC performs the exploration of paths. There are two common ways to perform the exploration of paths in EMC. The first alternative is based on *code re-execution*. In this approach, the model checker records the sequence of non-deterministic choices made along one execution path. The model checker explores another path by guiding the execution with a sequence of decisions derived from the exploration of some previous path. Another alternative is to make the model checker to *store the state* when performing a non-deterministic choice. Backtracking requires the model checker to restore the state that it previously stored. We focus our attention to model checker that implements path exploration by storing and restoring state.

One alternative to store and restore state is to use the support of the operating system. CMC [MPC+02], for example, a model checker written for the C language, forks the execution in non-deterministic choices, effectively enforcing a copy-on-write discipline on the data that several executions share. Another alternative is to make the state visible to the model checker, i.e.,

to enable the model checker to store, and restore the state. We focus our attention on such kind of model checkers that can manipulate the state. Examples of model checkers that operate in this manner include BogorVM [RDH03], JPF [LV01, VHB+03], and SpecExplorer [VCST05].

Path pruning (state comparison)

Several techniques exist to prune paths from the state space exploration. (See Section 5.1.1.) The techniques this dissertation proposes concentrate in pruning of paths based on isomorphism: the model checker prunes a path when it leads to a states that is isomorphic to one that it has already seen in previous explorations.

Typically, a model checker compares state by first computing a linear representation of state, then computing a hash from the linear representation, and finally using the hash to lookup or add the state in a hash table.

A model checker can compute a linear representation of the state, also called a *linearization*, serialization, or marshalling [Ios01, LV01], by traversing the root of the heap in a depth-first order. Two heaps are isomorphic if and only if their linearizations are equal.

2.6   Java PathFinder

We briefly discuss some concepts of JPF relevant for the techniques. More details on JPF can be found elsewhere [LV01, VHB+03].

JPF is a general-purpose explicit-state model checker for Java bytecodes. It takes as input a Java program and an optional bound on the length of the program execution and explores, up to the given bound, all executions that the program can have due to different thread interleavings and nondeterministic choices. JPF can generate as output those executions that violate a given property, for example, violate a state assertion. JPF can also generate as output test inputs for the subject program [VPK04, VPP05]. The main difference between JPF and a regular Java Virtual Machine (JVM) is that JPF can backtrack the program execution to a previously visited choice point and compare arbitrary JVM states.

JPF implements a special Java Virtual Machine (JVM) on top of a regular JVM. The main difference between JPF and a regular JVM is that it can backtrack the program execution by

restoring the state it previously encountered during the execution. Backtracking allows exploration of different executions from the same state. JPF implements backtracking by storing and restoring the state, and it represents this state in a special form (See Section 2.5).

2.6.1 JPF's representation of state

To efficiently perform the operations listed in Section 2.5, JPF uses a *special representation of the state.* This representation allows efficient storage, reconstruction, and comparison of states. This is possible because the state is *visible* to the model checker. The model checker can read and write to it directly. For example, the model checker can reconstruct only the portion of the state that actually changed across multiple non-deterministic choices.
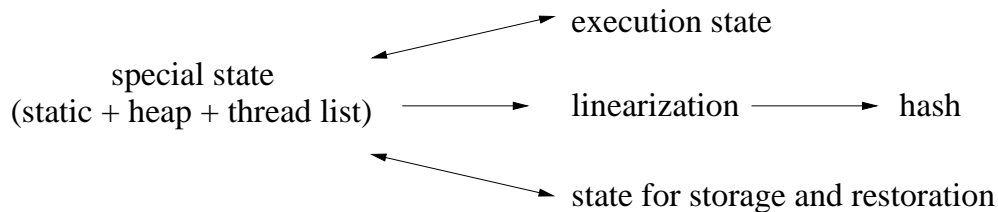
Figure 2.3: Flow of data between various state representations in EMC (JPF).

In practice, JPF keeps multiple representations of the state. Figure 2.3 relates these multiple representations of state. This figure reflects the design of JPF, but generalizes for other model checkers that use special state representation. The direction of the arrow indicates flow of data. The special state includes the heap, static area, and threads and provides the building blocks for the other specific representations. JPF, for example, internally represents an object with an array of integers. It represents the heap with an array of integer arrays. The execution state consists of a set of objects that JPF uses to interface with the special state. These data-structures read and update the special representation during execution.

The storage and restoration state is a compact representation of the fragments of the special state that allows fast reconstruction [VHBP00]. This representation reads and updates the special state representation: it stores the state in the event of non-deterministic choices, and restores the state when backtracking.

The representation called the *state linearization* [Ios01] enables comparison of state during the exploration. Linearization is the process of building state linearizations. It strives for building a

29

```
public class BST {
    private Node root;
    private int size;
    public void add(int info) { ... }
    public boolean remove(int info) { ... }
    ...
}
class Node {
  Node left, right;
  int info;
}
```

Figure 2.4: Fragment of the binary search tree appearing in Figure 1.1.

canonical representation of the state that the model checker can use for accurate comparison. (The model checker can efficiently build a linearization from the special state when objects already have linear representation in the special state representation. See Section 2.6.2) The model checker can produce a *state hash* from the linearization, and use hash sets to efficiently check whether a state has been seen before during the exploration.

2.6.2 Heap Representation

This section describes how JPF represents state internally. More specifically, we focus on how it represents the heap. While JPF also represents stack, thread information, class information, and all other parts of a JVM state, our techniques require only manipulations of the heap.

Each Java heap consists of a set of objects and some values for the fields of these objects. Each object has an identity, and each field has a type that can be either primitive (`int`, `boolean`, `float`, etc.) or a reference to another object (which can hold the special value `null`). JPF uses Java integers to represent object identifiers. JPF also uses Java integers to encode all field values, be they primitive or pointers. (JPF determines the meaning of various integers based on the field types kept in the class information.) Conceptually, JPF represents each object as an integer array, and the entire heap as an array of integer arrays.

Figure 1.1 shows a binary search tree class that implements a set. Each `BST` object stores the size of the tree and its root node, and each `Node` object stores references to the two children and an integer value. The `BST` class has methods to add and remove tree elements.

Figure 2.5 shows the representations of one example BST object in both JVM, as an object

30
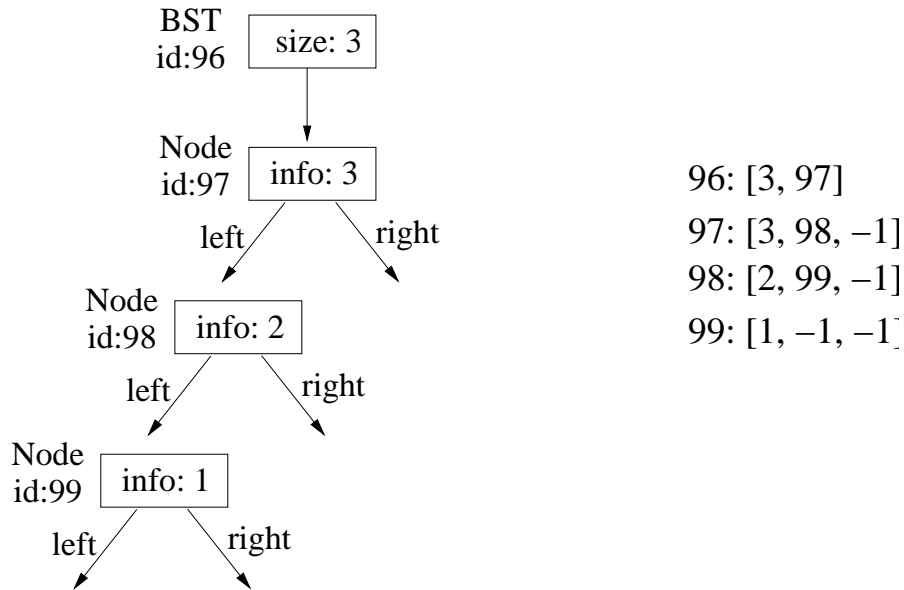
Figure 2.5: An example BST as an object graph and in the JPF heap representation.

graph, and in JPF, as an array of integer arrays. This example BST object can result from the sequence BST t = new BST(); t.add(3); t.add(2); t.add(1). This figure shows for each object its type, integer identifier, and the values of primitive fields. The reference fields not shown in the graph have the value null, represented as -1 in JPF.

2.6.3 Model Java Interface

This section describes the Model Java Interface (MJI), an existing mechanism in JPF for accessing the JPF state from the host JVM.

MJI enables delegation of JPF execution to the host JVM. MJI is analogous to the Java Native Interface (JNI) [jni] that allows the JVM to delegate execution to native code, written in the C language, for instance. MJI, like JNI, splits executions at the method granularity: the user can mark each method for execution in either JPF or in the host JVM. (JPF uses special name mangling to mark methods for the host JVM execution.) MJI also provides API that allows the host JVM execution to manipulate the JPF state representation, for example to read or write field values or to create new objects.

The libraries distributed with JPF use MJI to implement several parts of the standard Java library. The user can employ MJI to implement functionality that either requires higher perfor-

mance (e.g., copy of arrays) or is not available at the target level (e.g., reflection [SM03] in Java). MJI enables higher performance as it makes possible the use of native implementation and provides a mechanism to extend the support of features that are unavailable on the target level. JPF uses, specifically, MJI to implement several classes and methods from the `java.io` and `java.lang` packages.

2.7   The AODV case study

We next present Ad-Hoc On-Demand Distance Vector (AODV) routing [PR99], a widely used network protocol for wireless multi-hop ad hoc networks. We use AODV as a case study to evaluate the techniques this dissertation describes.

We consider an implementation of AODV based on the AODV Draft (version 11) [PR99] and implemented in J-Sim [J-S, Tya02], a network simulator written entirely in Java. AODV is a fairly complex network protocol whose J-Sim implementation (not including the J-Sim library) has about 1200 lines of code. This case study was used previously to evaluate a model checker specialized for J-Sim [SVH04, SVMH05].

An ad hoc network is a wireless network that comes together when and where needed, as a collection of wireless nodes, without relying on any assistance from an existing network infrastructure such as base stations or routers. Due to the lack of complete connectivity and routers, the nodes are designed to serve as routers (i.e., relays) and assist each other in delivering data packets.

The AODV State

An AODV network consists of several AODV nodes. Each node, n, contains among others, the following fields:

- $seqno_n$ is the identifier for this node

- bid_n is a sequential number that identifies the broadcast messages this node sends

- $bidcache_n$ is a set including a history of packet identifiers that this node has already processed

- $table_n$ is a routing table that stores information of the route to deliver messages.

In AODV, each node n in the network maintains a routing table with a set of entries. Each entry includes information about the route to a specific destination. A routing table entry (RTE) at node n to a destination node d contains, among others, the following fields:

- $nexthop_{n,d}$ denotes the address of the node to which n forwards packets to d

- $hops_{n,d}$ denotes the number of hops needed to reach d from n

- $seqno_{n,d}$ is a sequential used as a measure of freshness for this route

- $lifetime_{n,d}$ denotes the lifetime for this entry

Events

We use the term event to refer to an event that an environment, possibly external to the node, sends to the node. An event triggers an action. We list, in the following, important events that such environment sends to the nodes and also describe the actions the nodes take in response to these events.

- **Route from node n to d timeouts**. Periodically, the protocol sends a route timeout event to invalidate (but not delete) all RTEs that have not been used (e.g., to send or forward packets to the destination) for a certain duration; the AODV node uses the field *lifetime* of an RTE to determine whether it should invalidate the entry. Invalidating a RTE involves incrementing $seqno_{n,d}$ and setting $hops_{n,d}$ to ∞.

- **Node n requests a route to destination d**. If the node n has a valid route to d it responds with the route. Otherwise, it first creates an invalid RTE to d with $hops_{n,d}$ set to ∞. The node then increments bid_n and *broadcasts* a route request (RREQ) packet containing the fields $\langle n, seqno_n, bid_n, d, seqno_{n,d}, hopCount_q \rangle$. The $hopCount_q$ field is initialized to 1. The fields n and bid_n together identify a RREQ packet.

- **Delivering an RREQ packet that node n sends at node m**. Each node m, receiving the RREQ packet from node n, stores the pair $\langle n, bid_n \rangle$ in the field $bidcache_n$, so that m can later check if it has already received a RREQ with the same source address and broadcast ID

(this can happen in effect of multiple broadcasts). In the case the message is repeated, the incoming RREQ packet is discarded. In the case it is the first time this message is processed, m either satisfies or not the RREQ: it satisfies if n has a fresh enough route to m: node m compares the field $seqno_{m,d}$ with the field $seqno_{n,d}$ that the packet RREQ includes. In the case m satisfies the request, it sends a route reply packet (RREP) directly to d (using unicast). In the case node m cannot satisfy the request, it rebroadcasts the packet to its neighbors after incrementing the $hopCount_q$ in the packet.

- **Broadcast ID timeout at node** n. Each entry in the broadcast ID cache has a lifetime. Periodically, a broadcast ID timeout event is triggered causing the deletion of cache entries that have expired.

The RREP packet that a node m sends includes, among others, the fields $seqno_{m,d}$ and $hopCount_d$ denoting, respectively, the freshness of the route that node m offers and the number of hops between the node that receives this packet and d. If node m offers node n a new route to d, n compares $seqno_{m,d}$ (the destination sequence number of the offered route) to $seqno_{n,d}$ (the destination sequence number of the current route), and accepts the route with the greater sequence number. If the sequence numbers are equal, the offered route is accepted only if it has a smaller hop count than the hop count in the RTE; i.e., $hops_{n,d} > hops_{m,d}$.

Safety property

An important safety property in a routing protocol such as AODV is the *loop-freedom* property. Intuitively, a node must not appear more than once in a routing path; therefore, at each hop along a path from a node n to a destination d, either the destination sequence number must increase or the hop count must decrease. Formally, consider two nodes n and m such that m is the next hop of n to some destination d; i.e., $nexthop_{n,d} = m$. The loop-free property can be expressed as follows [BOG02, MPC$^+$02, TSH05, SVMH05]:

$$seqno_{n,d} < seqno_{m,d} \lor (seqno_{n,d} = seqno_{m,d} \land hops_{n,d} > hops_{m,d})$$

Chapter 3

Mixed Execution

This chapter presents *mixed execution,* a technique that can reduce overall exploration time in explicit-state model checkers that use special state representation.

Recall from Section 2.5 that explicit-state model checking of programs include three major operations: execution, path exploration, and path pruning. Execution of transitions refers to the operation that performs a deterministic step in the subject program, path exploration refers to the operation that performs exploration of *all* program paths created with non-deterministic choices, and path pruning refers to the operation that performs pruning of some of these paths based on some notion of equivalence, e.g., isomorphism of states that the model checker visits during the exploration. One possible way to implement these operations makes use of a *special state representation.* In this approach, execution operates using state that the model checker directly manipulates, path exploration stores and reconstructs such states at non-deterministic choice points, and path pruning can perform, for instance, by comparing states: the model checker identifies as equal two states visited in the exploration and prunes one of the two paths that leads to that state.

On the one hand, such approach allows efficient reconstruction and comparison of states as the model checker is able to directly manipulate the state. On the other hand, it sacrifices execution time: every read and write needs to access such special state (see Chapter 1). Writes, in particular, require the model checker to save information that enables restoration. In summary, the special representation makes the exploration of *all* executions efficient, although it makes *each individual* execution inefficient. Examples of model checkers that use special state representation include AsmLT [Fou], BogorVM [RDH03], JPF [LV01, VHB$^+$03], and SpecExplorer [VCST05]. We focus on such kind of model checkers.

The main idea of mixed execution is to execute *some* parts of the program with other, more

efficient, representation of state. The goal of the technique is to *reduce overall exploration time in model checkers that use special state representation.*

We use the term *deterministic blocks* to refer to blocks whose execution cannot make non-deterministic choices, e.g., due to thread interleavings or choices that the programmer may explicitly introduce. Mixed execution works by changing the representation of state at the boundaries of deterministic blocks. Within these boundaries, the model checker uses a representation more efficient for execution; it does not "track" the intermediate state changes that occur along the execution of sequential code.

We apply mixed execution on object-oriented programs and, in this context, deterministic blocks will correspond to bodies of sequential methods. In principle, however, one can apply the technique to any arbitrary block of deterministic code.

The technique applies the following steps to execute a method invocation: it translates the inputs of the method call (receiver and arguments) to a representation that favors execution, then executes the body of that method, and finally translates back the resulting state to the representation the model checker uses for regular execution, path exploration, and path pruning.

With mixed execution, the model checker still executes the other parts of the program and stores, restores, and compares the states as usual. It is important to point that mixed execution only reduces the execution time of deterministic blocks; it does not affect the order of exploration, the number of explored states, or any other aspect of the state exploration. In fact, the model checker can apply mixed execution in conjunction with techniques that improve orthogonal aspects of the exploration.

We evaluate mixed execution in the JPF model checker. Although we present the technique in the context of JPF, our main idea – executing parts of model checking on different state representations – generalizes to other model checkers that use some special state representation. These checkers do not need to be for Java or even based on virtual machines.

We evaluate mixed execution and an optimization of our baseline implementation on seven subject programs that use JPF to generate tests for data structures and on a larger case study. The experimental results show that mixed execution can improve the overall time for state exploration in JPF from 1.01x to 1.73x (with median 1.13x), while improving the time for the execution

```
public static void main(String[] args) {
  int M = Integer.parseInt(args[0]); // length of the sequence
  int N = Integer.parseInt(args[1]); // range of inputs
  main(M, N);
}

public static void main(int M, int N) {
  BST bst = new BST(); // empty tree
  for (int i = 0; i < M; i++) {
    int methNum = Verify.getInt(0, 2);
    int value = Verify.getInt(0, N-1);
    Verify.beginAtomic(); // begins deterministic block
    switch (methNum) {
      case 0: bst.add(value); break;
      case 1: bst.remove(value); break;
    }
    Verify.endAtomic();    // ends deterministic block
    stopIfVisited(bst);
  }
}
```

Figure 3.1: Driver for bounded-exhaustive exploration of the BST subject.

of deterministic blocks from 0.91x to 3.05x (with median 1.64x). Additionally, the optimization improves the baseline mixed execution implementation from 1.03x to 1.35x (with median 1.08x).

We will use the term *mixed mode* to refer to the mode of execution where the model checker changes state representation on the boundaries of a method call, and the term *native representation* to refer to a representation of state that favors execution. This representation is typically native to some environment. In the case of Java, for instance, native representation refers to the representation of state that the JVM uses.

Next, we present an example that highlights the key aspects in mixed execution. Then, we present more details of the technique, its implementation in JPF, and the experimental results we obtain. Finally, we discuss limitations of our technique and our current implementation as well as future work.

3.1 Example

This section presents an example that illustrates how mixed execution can speed up state exploration.

Figure 3.1 shows the method main(int M, int N) that expresses a driver similar to the one in

Figure 1.4. The model checker uses this driver to explore the state space of the `BST` class that appears in Section 1.3.1. The model checker explores all sequences of `add` and `remove` operations up to the bound `M` for for the sequence length and the bound `N` for the range of input values. The library method `Verify.getInt(int lo, int hi)` non-deterministically chooses a number between `lo` and `hi` (including `lo` and `hi`). The library methods `beginAtomic` and `endAtomic` mark a deterministic block[1]; these manually added annotations instruct the model checker to ignore non-deterministic choices within a given block. The goal of the driver is to explore different trees that can arise during the executions. The driver uses *state matching* [VPP05, VPP06, XMN04] to compare only the state of the tree, which the reference in `bst` denotes. If the state has been already visited during the exploration, the library method `stopIfVisited` instructs the model checker to backtrack the execution.

Without mixed execution, the model checker executes `add`, and `remove` methods using a special representation of state, which slows down every field read and write during the execution of method calls. Note, however, that to perform comparison and backtracking the model checker requires to access the state of the tree *only* at the beginning and at the end of these methods as they are deterministic.

We now highlight the key steps for executing the method calls to `add` and `remove` in mixed mode.

- At the beginning of each deterministic method invocation, mixed execution translates all the objects reachable from the method parameters, including the tree reachable from `this`, from the special representation into the native representation.

- Mixed execution then invokes the method on the tree using native state representation. The method execution can modify state.

- At the end of each method execution, mixed execution translates the state back from native representation into the special representation. The model checker then resumes the exploration that operates on the special state.

---

[1]Prior to its version 4, JPF allowed explicit non-deterministic choices introduced by the programmer, say via `Verify.getInt()`, to appear inside atomic blocks. We run mixed execution on version 3.1.2 but do *not* introduce any non-deterministic choice inside these blocks.

The speedup (or slowdown) that mixed execution achieves depends on the cost of performing the translations and the length of the method execution. The smaller the state is, the less mixed execution has to copy during state translations. Also, the longer the execution is, the more the model checker saves by executing with native states.

The baseline implementation of mixed execution copies the entire state to native representation regardless of the amount of state the execution actually reaches. For this reason, we call it *eager translation*. We optimized the baseline implementation to perform the translation only when execution reads a field. We call this form of translation *lazy*.

In our running example with BST, the results vary with the value for the bounds M and N. We set M = N in all experiments, and the value ranges from 5 to 10, as done in previous studies on testing [VPP05, VPP06, XMN04]. For these bounds, JPF with mixed execution (and lazy translation) takes from 1.04x to 1.22x less time for overall state exploration than JPF without mixed execution. Considering only the executions of add and remove methods, mixed execution provides from 1.43x to 1.53x speedup.

It is important to point that the state space exploration includes, besides execution, path exploration and pruning. Mixed execution only reduces the method execution time, which can be significantly high, while the cost of the rest of the state exploration remains the same.

3.2 Technique

We next present in more detail the operations involved in mixed execution. We describe the pseudo-code of a procedure that intercepts method invocations in an explicit-state model checker for object-oriented programs. The procedure that appears in Figure 3.2 shows the essential parts of our technique.

We assume that the methods on which the model checker applies mixed execution are deterministic, i.e., execution makes no thread-interleavings or user-provided non-deterministic choices. We also assume that the model checker uses special state representation. This is key to obtaining execution speed up.

The procedure invokeMethod takes as input a method name and a list of arguments and produces as output the state that results from the method execution. We use an array of integers to identify

```
void invokeMethod(Method m, int[] args) {
  if (m.shouldBeExecutedOnNative()) {
    // translate arguments to native form
    Object[] inputs = translateSpecialToNative(args);
    Throwable throwable = null;
    try { // uses reflection to invoke the method
      Object result = m.invoke(inputs);
    } catch (Throwable t) {
      throwable = t;
    } finally { // executes regardless of exception
      // translate arguments back to the special representation
      translateNativeToSpecial(inputs);
      if (throwable == null) {
        // translate the return value
        int result = translateNativeToSpecial(result);
        pushOnStack(result);
      } else { // method call raised an exception
        // translate the exception
        int specialThrowable = translateNativeToSpecial(throwable);
        raiseException(specialThrowable);
      }
    }
  } else { /* regular invocation */ ... }
}
```

Figure 3.2: Pseudo-code of the method invocation in model checker using mixed execution.

the arguments of the method call. As we showed in Section 2.6, JPF identifies an object with an integer. The method signature distinguishes between the use of an integer to encode an object or a primitive. Important to note is that the method execution operates with a state representation different than the one that the model checker uses.

Mixed execution associates to each method a flag indicating whether or not it should execute in mixed mode, i.e., whether or not it should execute the method outside the control of the model checker. If the model checker makes a call to a method that has the flag set, the procedure `invokeMethod` translates the state of all method arguments, including the receiver, to a native form. The method `translateSpecialToNative` performs such translation. It then executes the method. Note that the inputs to the method call have `Object` type instead of integer. Finally, the procedure translates the state the method execution produces or modifies back to the representation the model checker uses. The method `translateNativeToSpecial` performs such translation.

The procedure handles the cases when the method execution returns normally as well as the case when it throws an exception. When the method execution returns normally, the procedure translates the resulting state to the model checker representation and pushes it on the stack. When

the method execution raises an exception, the procedure catches the exception and indicates that to the model checker. In either case (i.e., regular or exceptional execution), mixed execution translates the state of the input arguments back to the model checker special representation since execution can mutate any part of the state reachable from the method arguments.

Note that the arguments represent roots for the part of the heap that the method can manipulate. The heap may be much larger than the part reachable from the roots, but the method cannot manipulate the part that is not reachable from the roots. In general, however, the roots should also include static fields.

3.3   Implementation

This section details the `invokeMethod` procedure that Figure 3.2 describes in the context of JPF. We first provide additional background on JPF that is necessary for the discussion, then we describe different implementations of the algorithm that translates rooted heaps in JPF: `translateSpecialToNative` and `translateNativeToSpecial`.

We use the term *JPF state* to refer to the special representation of the program state that JPF internally uses, and the term *JVM state* to refer to the native representation we use to speed up execution in JPF. JVM state corresponds, as we mentioned before, to the state of the JVM that actually hosts JPF.

3.3.1   Model Java Interface

We implement mixed execution by modifying the source code of JPF. Our implementation uses, in a novel way, a mechanism that already exists in JPF; to quote from the JPF manual [MVP]:

> Host VM Execution - JPF is a JVM that is written in Java, i.e. it runs on top of a host VM. For components that are not property-relevant, it makes sense to delegate the execution from the state-tracked JPF into the non-state tracked host VM. The corresponding Model Java Interface (MJI) mechanism is especially suitable to handle IO simulaion [sic] and other standard library functionality.

```
void jpfInvoke(Method m, int[] args) {
  if (m.shouldBeExecutedOnJVM()) {
    // get the JPF execution environment and
    // translate arguments from JPF to JVM
    MJIEnv env = JPF.getMJIEnv();
    Object[] inputs = translateJPF2JVM(env, args);
    Throwable throwable = null;
    try {
      // use reflection to invoke the method on JVM,
      // giving it the translated values as the arguments
      Object result = m.invoke(inputs);
    } catch (Throwable t) {
    } finally {
      // translate the heap reachable from the roots from JVM to JPF
      translateJVM2JPF(env, inputs);
      if (throwable == null) {
        // translate the return value
        int jpfResult = translateObjectJVM2JPF(env, result);
        MJI.pushOnStack(jpfResult);
      } else { // method call raised an exception
        // translate the exception
        int jpfThrowable = translateObjectJVM2JPF(env, t);
        MJI.raiseJPFException(jpfThrowable);
      }
    }
  } else { /* regular invocation */ ... }
}
```

Figure 3.3: Pseudo-code of the method invocation for the host JVM execution.

MJI is an API that allows the host JVM to manipulate JPF state. The novelty of mixed execution is the use of MJI to delegate execution from a state representation that JPF tracks into a native representation (which JPF does not track); this translation is valid even for blocks of code that are property-relevant. Mixed execution executes on the host JVM some program code that can modify the program state and thus affect a property, for example, part of the state that an assertion reads. As a matter of fact, we use our technique in the execution of property-relevant blocks during the model checking of a network protocol. Previous uses of MJI in JPF did not execute such program code on JVM and did not translate the state between JPF and JVM representations.

Figure 3.3 specializes the `invokeMethod` procedure appearing in Figure 3.2 to JPF. The method `jpfInvoke` uses MJI to read from and write to the JPF state; the variable `env` provides access to the MJI API. The method `translateJPF2JVM` translates the state from the JPF to the JVM representation and the method `translateJVM2JPF` performs the inverse translation. Note that these

methods read and write to the JPF heap via the MJI API, which is accessible via variable `env`.

The baseline translation algorithm always translate at the beginning of a deterministic block the entire state reachable from a set of roots from the JPF to the JVM representation. Note that even this state can be a tiny part of the entire JVM state. We call the baseline translation *eager*. Effectively, the eager mixed execution translates the entire state that any execution of the deterministic block *may* read or write. In contrast, *lazy translation* starts the execution without translation and then, during the execution, translates on demand those state parts that the specific execution *does* read or write. As a result, lazy mixed execution performs less translation and can speed up the eager mixed execution.

3.3.2 Eager Translation

Figure 3.4 shows the pseudo-code of the method that translates the state from JPF to JVM. The inputs to the method are an `MJIEnv` object, which encodes the entire environment/state of the JPF execution, and an array of method arguments, encoded in JPF as integers (see Section 2). (For instance methods, the first argument represents `this`.) The output of the method is an array of JVM objects that correspond to the arguments. The method uses a depth-first traversal of the JPF heap reachable from `args` to create an *isomorphic* JVM heap [MK01, BKM02]. The method creates two maps that keep the correspondence between the JPF and JVM object identities. These maps initially start empty, but the helper method adds for each JPF object an appropriate JVM object. The method uses the map from JPF to JVM to handle heap aliases. (The use of the map also ensures that the translation terminates when the heap has cycles.) The map from JVM to JPF will be used during the translation at the end of the execution. The method and the helper use several MJI calls (on the `env` objects) to get the values of fields and to get the types of the arguments and fields.

Figure 3.5 shows the pseudo-code of the method that translates the state from JVM to JPF. The inputs to the method are an `MJIEnv` object and an array of the inputs, which represent the roots of the heap at the beginning of the execution. The effect of the method is to update the JPF state. The method uses a depth-first traversal of the JVM heap reachable from the `inputs` roots to appropriately update the JPF heap to be isomorphic to the corresponding JVM heap. The

```
Map<int, Object> mapJPF2JVM;
Map<Object, int> mapJVM2JPF;
// main method that translates all arguments in the pre-state
Object[] translateJPF2JVM(MJIEnv env, int[] args) {
  mapJPF2JVM = new Map<int, Object>();
  mapJVM2JPF = new Map<Object, int>();
  Object[] result = new Object[args.length];
  for (int i = 0; i < args.length; i++) {
    Type t = env.typeOf(args[i]);
    if (t.isPrimitive()){ result[i] = correspondingPrimitiveObject(t,args[i]); }
    else{ result[i] = translateObjectJPF2JVM(env, args[i]); }
  }
  return result;
}
// helper method that translates all fields reachable from a reference
Object translateObjectJPF2JVM(MJIEnv env, int jpfPointer) {
  if (jpfPointer == MJIEnv.NULL) return null;
  if (mapJPF2JVM.contains(jpfPointer)) return mapJPF2JVM.get(jpfPointer);
  // create a new object
  Object o = translateOneReferenceJPF2JVM(env, jpfPointer);
  // set the fields of the object recursively
  foreach (field f in o.getFields()) {
    int value = env.getFieldValue(jpfPointer, f);
    Type t = env.typeOf(f);
    if (t.isPrimitive()){ setField(o, f, correspondingPrimitiveObject(t, value)); }
    else{ setField(o, f, translateObjectJPF2JVM(env, value)); }
  }
  return o; // return the new object with all fields translated
}
// helper method that translates only one reference
Object translateOneReferenceJPF2JVM(MJIEnv env, int jpfPointer) {
  if (jpfPointer == MJIEnv.NULL) return null;
  if (mapJPF2JVM.contains(jpfPointer)) return mapJPF2JVM.get(jpfPointer);
  // get the type of JPF object "jpfPointer"
  Class c = env.getClass(jpfPointer);
  // create a new object of class "c" using reflection
  Object o = c.newInstance();
  // update the mappings between JPF and JVM objects
  mapJPF2JVM.put(jpfPointer, o);
  mapJVM2JPF.put(o, jpfPointer);
  return o;
}
```

Figure 3.4: Pseudo-code of the algorithm that translates the state from JPF to JVM.

traversals keep the set of visited objects. It is important to distinguish this set and the map from

JVM to JPF objects. In the translation from JPF to JVM, a map is used both to keep track of

visited (JPF) objects and to provide the mapping of identities. However, in the translation from

JVM to JPF, a map is only used to provide the mapping of identities, because an object should

be traversed even if it is in the map. Moreover, the translation must preserve the original JPF

identity of nodes. The translation method and its helper use several MJI calls (on the env objects)

44

```
Set<Object> visited;
// main method that translates the post-state
void translateJVM2JPF(MJIEnv env, Object[] inputs) {
  visited = new Set<Object>();
  for (int i = 0; i < inputs.length; i++) {
    if (!(env.typeOf(inputs[i]).isPrimitive())) {
      translateObjectJVM2JPF(env, inputs[i]);
    }
  }
}
// helper method that translates one object
int translateObjectJVM2JPF(MJIEnv env, Object o) {
  if (o == null) return MJIEnv.NULL;
  if (!visited.contains(o)) {
    visited.add(o);
    // get type of the object
    Class c = o.getClass();
    // get (or create if necessary) the corresponding JPF object
    int jpfPointer;
    if (!mapJVM2JPF.contains(o)) {
      // create new JPF object of the same type
      jpfPointer = env.createNewObject(c);
      mapJVM2JPF.add(o, jpfPointer);
    } else {
      jpfPointer = mapJVM2JPF.get(o);
    }
    // set the fields of the object recursively
    foreach (field f in c.getFields()) {
      // use reflection to get the field value
      Object value = f.getFieldValue(o);
      Type t = f.getType();
      if (t.isPrimitive()) {
        env.setFieldValue(jpfPointer, f, correspondingPrimitiveJPF(t, value));
      } else {
        env.setFieldValue(jpfPointer, f, translateObjectJVM2JPF(env, value));
      }
    }
  }
  return mapJVM2JPF(o);
}
```

Figure 3.5: Pseudo-code of the algorithm that translates the state from JVM to JPF.

to create new objects and set the values of fields.

3.3.3 Lazy Translation

Lazy translation is an optimization of the baseline translation that only perform the translation
of the parts of the heap that a method execution actually needs during execution. While eager
translation translates the entire heap at the beginning of the execution, the lazy algorithm translates
only the arguments and not all fields reachable from them. During the execution, however, lazy

translation performs a check for each field read and write to determine whether the field has been translated from JPF to JVM. If not, lazy translation translates only that one field and continues the execution. By the end of the execution, lazy translation typically translates into JVM only a small part of the heap reachable from the method arguments at the beginning.

Lazy translation requires some changes to the code of the methods executed by mixed execution. Specifically, lazy translation requires the checks for each field read and write. We achieve those checks using *code instrumentation*. Figure 3.6 shows a part of the code from the `TreeMap` example before and after instrumentation. For each field, the instrumentation adds (i) a boolean flag that tracks whether the field has been translated from JPF to JVM, (ii) a method for reading the field value (translating it from JPF, if necessary), and (iii) a method for writing the field value. The instrumentation also replaces all field reads and writes in the original code with the invocations of appropriate methods. Finally, the instrumentation adds a special constructor to create objects without setting the flags. A similar instrumentation has been used previously in testing and model checking [BKM02, VPK04].

At the end of a method execution on the host JVM, mixed execution with lazy translation traverses the JVM heap similarly as mixed execution with eager translation. In contrast to eager translation, however, only those fields whose flags are set to `true` are translated from JVM to JPF and recursively followed further. A further optimization would be to have "dirty flags" to avoid translation from JVM to JPF for the fields whose value was not changed.

3.4   Evaluation

We next discuss the experiments used to evaluate mixed execution. We have implemented mixed execution by modifying the JPF code [MVP] to include the algorithms from figures 3.3, 3.4, and 3.5. We have also implemented a prototype tool that automates instrumentation for lazy translation as shown in Figure 3.6.

We evaluate mixed execution on six subject programs that use JPF for state exploration in data structures. We also evaluate mixed execution on a network protocol for which JPF finds an injected error. The blocks of code delegated to mixed execution are deterministic: they are sequential code without manually inserted non-deterministic choices (`Verify.getInt` calls).

```
// Original code, before instrumentation.
public class TreeMap {
  static class Entry {
    Entry left;
      ...
    }
  public Object put(Object key, Object value) {
    ... = e.left; // field read
    e.left = ...; // field write
  } ...
}

// Code after instrumentation.
public class TreeMap {
  static class Entry {
    Entry left;
    boolean _mixed_is_copied_left = false;
    Entry _mixed_get_left() {
      if (!_mixed_is_copied_left) {
        MJIEnv env = JPF.getMJIEnv();
        int jpfPointer = env.getFieldValue(mapJVM2JPF(this), "left");
        left = translateOneReferenceJPF2JVM(env, jpfPointer);
        _mixed_is_copied_left = true;
      }
      return left;
    }
    void _mixed_set_left(Entry e) {
      left = e;
      _mixed_is_copied_left = true;
    }
    ...
  }
  public Object put(Object key, Object value) {
    ... = e._mixed_get_left(); // field read
    e._mixed_set_left(...); // field write
  } ...
}
```

Figure 3.6: Example code before and after instrumentation.

We performed all experiments on a Pentium 4 3.4GHz workstation running under RedHat Enterprise Linux 4. We used Sun's 1.4 JVM, allocating 1.5 GB for the maximum heap size, and JPF version 3.1.2. We compare the time that JPF takes for exploration with and without mixed execution. In both cases, we set JPF to use breadth-first state exploration. We also enable all JPF optimizations, including partial-order reductions [VHBP00], the use of MD5 hashing function [MVP], and the exact state comparison with respect to isomorphism [VPP05, VPP06].

3.4.1 Basic subjects

We evaluate mixed execution on the six data structures listed in Figure 3.1. We take the subjects from previous studies on model checking and testing:

- `bst` is our running example that implements a set using binary search trees [BKM02,XMSN05]

- `disjset` is an implementation of a union-find data structure implementing disjoint sets [XMN04]

- `linkedlist`, `treemap`, and `vector` are from the Java 1.4 Collection Framework.

- `trie` implements a dictionary, i.e., it stores a collection of strings sorted lexicographically [XMSN05]

- `ubstack` is an implementation of a stack bounded in size, storing integer objects without repetition [SLA02, XMN04, CS04, PE05]

Our state exploration considers the methods that add, remove, and search for elements in each data structure, as listed in Figure 3.1.

subject	methods explored
bst	add, remove
disjset	union, find
linkedlist	add, removeLast, contains
treemap	put, remove, get
trie	add, is_word, is_proper_prefix
ubstack	push, pop
vector	addElement, removeElement, elementAt

Table 3.1: Subjects used in the Mixed execution experiments.

Each experiment uses an execution driver similar to that in Figure 3.1. By default, we use mixed execution with lazy translation. Figure 3.2 tabulates the results. We set the same bounds for the method-sequence length and for the range of values. For each subject and several bounds, we tabulate the number of states that JPF explores (which is the same with or without mixed execution), the total number of bytecodes that JPF executes (with mixed execution, the host JVM executes some bytecodes), the overall time for exploration, and the time for execution of methods marked for mixed execution. All times are in milliseconds. The columns labeled *JPF* and *mixed* represent the runs of JPF without and with mixed execution, respectively. The *speedup* columns show the improvement that mixed execution provides.

experiment			# bytecodes		method exec. only			total time		
subject	N	# states	std.	mixed	std.	mixed	speedup	std.	mixed	speedup
bst	5	188	147670	17501	252	174	1.45x	1521	1468	1.04x
	6	731	804672	81559	763	499	1.53x	2699	2489	1.08x
	7	2950	4302804	388830	2872	2004	1.43x	7747	6722	1.15x
	8	12235	22671033	1867625	14340	9536	1.50x	32420	27262	1.19x
	9	51822	118130316	8999159	73605	48568	1.52x	155304	129815	1.20x
	10	223191	610379638	43444381	368394	246979	1.49x	775443	636310	1.22x
disjset	5	77	207261	21507	158	173	0.91x	1988	1961	1.01x
	6	516	2067901	161408	917	640	1.43x	7835	7222	1.08x
	7	4747	27152409	1874435	12114	6583	1.84x	77685	68629	1.13x
linkedlist	5	3906	302915	105135	719	647	1.11x	3528	3423	1.03x
	6	55987	4218362	1446824	8547	7729	1.11x	29408	28194	1.04x
	7	960800	70962589	24157789	147158	129656	1.13x	487105	465557	1.05x
treemap	5	72	92740	7586	274	167	1.64x	1421	1325	1.07x
	6	185	361600	25864	599	340	1.76x	1976	1775	1.11x
	7	537	1223256	79470	1458	790	1.85x	3419	2763	1.24x
	8	1613	4629574	277476	4671	2340	2.00x	8533	6083	1.40x
	9	4709	16681289	952976	16281	7556	2.15x	26575	17245	1.54x
	10	13189	54581750	3008954	60371	23500	2.57x	88914	51375	1.73x
trie	5	32	120869	4839	238	146	1.63x	1306	1261	1.04x
	6	64	293899	10855	350	251	1.39x	1614	1524	1.06x
	7	128	690129	24359	616	397	1.55x	2223	2021	1.10x
	8	256	1679127	54311	1150	624	1.84x	3851	3282	1.17x
	9	512	4018501	120103	2500	1171	2.13x	7466	6174	1.21x
	10	1024	9190465	263549	5328	2385	2.23x	15052	12346	1.22x
ubstack	5	616	181217	19677	391	232	1.69x	1807	1659	1.09x
	6	3865	1561823	132475	1511	577	2.62x	4522	3521	1.28x
	7	27455	14940706	1038230	11410	3740	3.05x	26475	17422	1.52x
vector	5	5042	892349	120244	806	428	1.88x	3633	3258	1.12x
	6	68781	13596654	1605126	10096	3601	2.80x	31989	25346	1.26x
	7	1140940	247372481	26241879	172468	58623	2.94x	954179	820633	1.16x

Table 3.2: Comparison of JPF without and with mixed execution.

experiment			method exec. only			total time		
name	N	# states	eager	lazy	speedup	eager	lazy	speedup
	4	25	127	89	1.43x	1158	1122	1.03x
	5	72	231	157	1.47x	1389	1327	1.05x
treemap	6	185	437	338	1.29x	1900	1776	1.07x
	7	537	1071	777	1.38x	3079	2758	1.12x
	8	1613	3687	2375	1.55x	7311	6136	1.19x
	9	4709	12164	7582	1.60x	22171	17313	1.28x
	10	13189	40834	24252	1.68x	68388	51669	1.32x
	4	16	151	86	1.76x	1120	1080	1.04x
	5	32	196	153	1.28x	1300	1260	1.03x
trie	6	64	291	226	1.29x	1602	1521	1.05x
	7	128	493	375	1.31x	2185	2014	1.08x
	8	256	1229	680	1.81x	3888	3281	1.19x
	9	512	3027	1189	2.55x	7917	6190	1.28x
	10	1024	6609	2270	2.91x	16590	12315	1.35x

Table 3.3: Comparison of eager and lazy translations for Mixed execution.

The results show that mixed execution can reduce the overall state exploration time from 1.01x to 1.73x (with median 1.13x) , while reducing the method execution time from 0.91x to 3.05x (with median 1.64x). Note that for very short executions (such as `DisjSet` for bound 5), mixed execution may actually slow down JPF as the overhead of translation outweighs the benefit of execution on the host JVM. As a matter of fact, for all subjects and very small bounds, mixed execution slows down JPF. However, the more important cases are when the execution is long. As the results show, the longer the execution gets, the more benefit mixed execution provides.

3.4.2   Comparison of Eager and Lazy translations

Figure 3.3 compares mixed execution with eager translation and mixed execution with lazy translation. For two subjects and several sizes, we tabulate the time for execution of methods marked for mixed execution and the overall time for state exploration.

Compared to eager translation, lazy translation reduces the overall time from 1.03x to 1.35x (with median 1.08x), while reducing the method execution time from 1.28x to 2.91x (with median 1.47x). Note again that the longer the execution gets, the more benefit lazy translation provides. But, in contrast to the previous experiment that compares mixed and regular execution, this experiment shows that lazy translation outperformed eager for all bounds.

3.4.3   AODV Case Study

We also evaluated mixed execution on a larger application, namely the implementation of the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [PR99] in the J-Sim network simulator [J-S]. This application was previously used to evaluate a J-Sim model checker [SVMH05].

AODV is a routing protocol for ad-hoc wireless networks, i.e., a wireless network where each node dynamically decides, based on its current local state, the destination of a packet. We use the term *AODV node*, or just AODV, to refer to both the network node and the software component that models that node. Each AODV node contains an identifier and a routing table that describes where a message should be delivered next, depending on the destination. The safety property we check in this study expresses that all routes from a source to a destination should be free of cycles, i.e., the same node cannot appear more than once in any route (see Chapter 2.7).

It is important to point out that we simplified the J-Sim simulator in order to conduct this study on JPF. We realized that J-Sim creates several threads to manage the simulation. In particular, J-Sim creates threads to deliver messages across different network nodes. In such scenario, JPF could not explore a large portion of the AODV state space. We re-implemented the communication layer the simulator uses. This allowed the exploration to scale better.

3.4.4   Test driver

As discussed in Chapter 1, a test driver is a component that constructs sequences of method invocations to exercise the subject under different scenarios. We use the term *event* to denote an event of the network protocol that the test driver emits by invoking a specific operation, e.g., a message loss or a route request. For each event we associate a function that indicates whether it is possible to invoke the event operation in a particular state. We call such function the *event guard*. It takes a state as input and returns a boolean. The driver produces an environment that executes all sequences of protocol events, satisfying the guards, up to a configurable bound on the sequence length. The list that follows describes the events and their corresponding guards.

- Route from node n to d timeouts. This event is enabled whenever node n has a valid routing table entry to destination d. The event invalidates the routing table entry.

experiment		#bytecodes		method exec. only			total time		
N	#states	std.	mixed	std.	mixed	speedup	std.	mixed	speedup
8	2295	24612410	17466413	9070	2770	3.27x	49740	43681	1.14x
9	3209	37172718	26750218	13765	3889	3.54x	74330	60382	1.23x
10	4338	54179173	39374489	24258	5575	4.35x	123920	87837	1.41x

Table 3.4: Model checking AODV without and with mixed execution.

- Node n requests a route to destination d: This event is enabled if the node does not have a valid entry in the routing table to the destination d. The event sends a broadcast message of type RREQ (stands for route request).

- Delivering an AODV packet that node n sends at node m. This event is enabled if the network contains at least one AODV packet such that n is the next hop towards the destination. The event removes this packet from the network and forwards it to node n. The AODV implementation at node n processes this packet accordingly.

- Broadcast ID timeout at node n. This event is enabled if there is at least one entry in the broadcast ID cache of node n (see Chapter 2). The event is handled by deleting these broadcast id entries.

- Restart of the AODV process at node n. This event is always enabled. This event reinitializes the state the AODV at node n.

- Loss of an AODV packet with node n as address. This event is enabled if the network contains at least one AODV packet that has node n as destination. The event removes this packet from the network.

3.4.5   Error injection and oracle

We consider an initial state of an ad hoc network consisting of K nodes: n_0, n_1, ..., n_{K-1}. The initial topology is a chain where each node is a neighbor of both the node to its left and the node to its right (if they exist). All i nodes, for $0 \leq i \leq K - 2$, have valid routing table entries to node n_{K-1}. We only consider n_{k-1} as the destination node.

We manually inject an error as follows. We change the procedure that processes event: timeout of the route to destination d at node n. Instead of *invalidating* the entry and keeping it in the
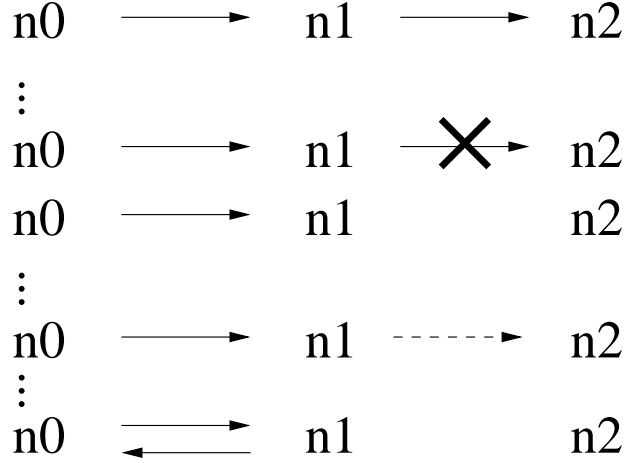
Figure 3.7: Sequence of AODV states leading to a bug. An arrow from a node to another indicates that the first can route a message to n_2 using the second. The dashed arrow denotes a dummy route created when n_1 sends a broadcast messages on request for a route to n_2.

table, the procedure *deletes* the entry to node n from the routing table. This change produces a different protocol behavior that results in a violation of the safety property.

Consider the following scenario. The network is initially a chain with $K = 3$, then a timeout event of the route n_1 to destination n_2 occurs, and later n_0 offers n_1 a route to n_2. Figure 3.7 illustrates the different states of the network according to this scenario up to a state that violates the safety property. The arrows denote the destination a message should take if a node needs to send/forward a packet to n_2. Initially, the network forms logically a chain of AODV nodes. Then the timeout event occurs; this event causes the removal of the route to n_2 from the routing table of n_1. Node n_1 will then request a route to node n_2 and when sending this event establishes the sequence number to -∞. Finally, node n_0 offers node n_1 a route to node $n2$ via itself. Node n_1 accepts this root because $seqno_{0,2} > seqno_{1,2}$. ($seqno_{1,2}$ corresponds to the dummy value node n_1 associates in its table when requested a route to n_2.) The acceptance of this route produces a states that violates the property of interest as the route from n_0 and n_1 to n_2 contain a cycle. (See Section 2.7)

The case of $K > 3$ is similar. The reader can find a detailed explanation of this injected error elsewhere [SVH04].

3.4.6 Mixed execution.

To apply mixed execution on AODV, we needed to determine which parts of the AODV code to execute on the host JVM. We first marked to run on mixed mode all methods of the data structures that AODV uses to represent protocol data. This includes routing tables and packet queues. Then, we profiled the model checker execution to find that it spends a lot of execution time in the methods from the library class `Port`. This class handles the sending and receiving of packets between network nodes [Tya02], so we also marked those methods for execution in mixed mode.

Figure 3.4 shows the improvements obtained with mixed execution on AODV. We tabulate, for a range of number of nodes and length of the event path, given by column N, the overall state-space exploration time and the method execution time. Mixed execution improves the overall exploration time from 1.14x to 1.41x (with median 1.23x), and the method execution time from 3.27x to 4.35x (with median 3.54x).

3.5 Limitations and Future Work

We now list the limitations of mixed execution (and its current implementation) as well as plans for future work.

Safe handling of atomicity. The technique and implementation takes as input a set of methods for execution in mixed mode. We assume that the input methods are deterministic. We do *not* currently provide a tool to *infer* deterministic blocks, and our implementation does not *check*, during execution, whether blocks are indeed deterministic. Inferring or checking deterministic blocks can help one to apply mixed execution. Extensive research has been done related to the detection of data-races and errors due to lack of atomicity in multi-threaded programs both statically [BLR02,FQ03] and dynamically [SBN+97,WS06]. Data races are concurrency errors that result from accesses to data, which different threads share, without adequate synchronization to protect such accesses. Atomicity is a stronger notion of concurrency; it imposes additional constraints on the interleavings so that executions only produce behaviors equivalent to those in which blocks declared atomic run sequentially. There can be program blocks which are free of data races but are not atomic [FQ03,WS06]. Mixed execution requires the user to annotate which input meth-

ods are atomic (deterministic) so that the model checker will not miss any behavior by imposing a sequential execution order. A very conservative approach to automate this check in mixed execution is to track for each thread the set of shared variables it can access. The model checker would then identify as potentially non-deterministic an execution of a method that can access data that another thread can read or write. We leave as future work to investigate approaches for identifying atomic blocks to speed up execution.

Improved translation from JVM to JPF state. The current implementation of Mixed execution writes back to the JPF state all fields that have been read during the execution of a method in mixed mode. We can improve the translation from JVM to JPF representation to avoid writing fields which are not *modified* in the execution. It is possible, for example, to associate to each field a dirty flag that indicates whether or not execution wrote to that field, or associate a dirty flag to each object.

Improved translations with memoization. Using immutable data is a common design pattern. For example, most classes in the `java.lang` package follow this pattern: it is not possible to mutate the state of a `String` or an `Integer` object. Mixed execution can also be improved by identifying types that are strictly functional or functional with respect to the environments a test driver exercises. The model checker can translate to native representation the objects of these types once and for all (modulo garbage collection). The native state could keep references to these objects in native representation to avoid repeated translations of state.

Support static fields. The current implementation of mixed execution does not consider static fields, i.e., it does not update the model checker's special state representation when code that executes in mixed mode writes to static fields. The experiments did not exercise such cases and thus static fields was not an issue in our evaluation. In principle, we can instrument all writes to non-final static fields to record modification. Mixed execution could therefore identify and mutate the parts of the state of the model checker that corresponds to static fields that have been modified during mixed execution.

Speeding up native method calls. The current implementation of Mixed execution uses Java reflection [SM03] to invoke operations on the subject under test. This invocation is slow. The implementation can avoid the cost of reflective calls by instrumenting specialized operations
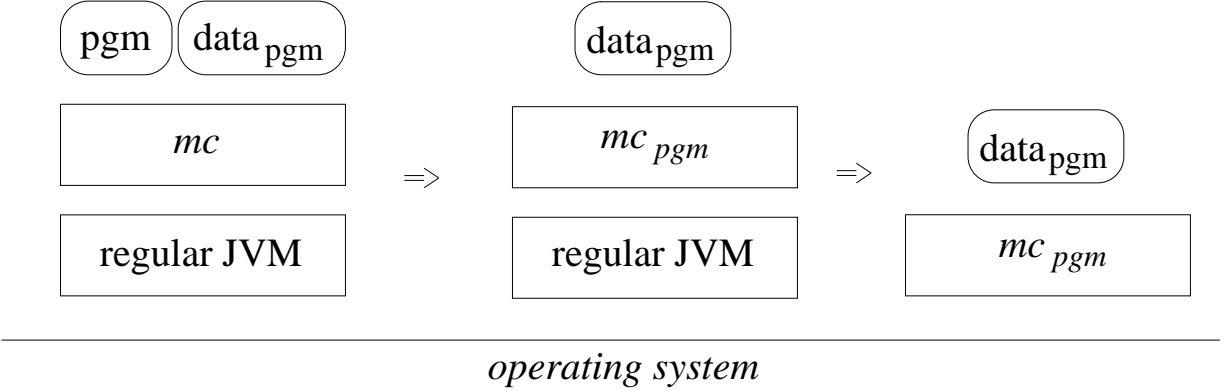
Figure 3.8: Model checkers with different environments for execution. The right arrows denote progressive program transformations. The leftmost model checker executes program transitions on the program mc, the second on the regular JVM, and the third directly on the host machine.

on each subject or even creating specialized bytecode instructions, in the case of JPF, for each method or class that runs in mixed mode.

Engineering a model checker for fast execution. We want to investigate different designs in the construction of a program model checker. In particular, we want to investigate designs of model checkers that can lead to efficient execution of transitions. Figure 3.8 illustrates possible alternatives for the construction of a model checker.

In the first alternative, the model checker is a Java program that runs on top of a JVM. The program to model check, pgm, is passed as data to the program mc that corresponds to the model checker. This results in slow execution: the bytecodes of the subject program executes in a virtual machine that runs on top of a regular virtual machine.

The second alternative specializes the model checker to one specific program. This avoids interpretation of the transitions of pgm in mc. Automatic transformation from the first to the second design requires a compiler, in essence a partial evaluator [JGS93], that takes the model checker mc as input and produces a new model checker specialized to the new program. More precisely, the compiler translates the program $mc : pgm \times data_{pgm} \rightarrow output$ into the program $mc_{pgm} : data_{pgm} \rightarrow output$. The input data refers to the parameters for the search and output refers to a description of the program paths that the model checker explores. Another approach to perform this transformation is to modify the input program to become model checkable. For example, instrumenting field writes to allow restoration of state and introducing non-deterministic

56

choices wherever is necessary.

The last design alternative illustrates a model checker that runs on top of the host machine. The program $mc_p gm$, combining model checker and input program, is written in C, for instance. Transformation of the second to the third design requires translation from Java to, say, C [jik].

JPF, for example, follows the first alternative. The JPF's JVM runs on top of a regular virtual machine, but the program to model check runs on top of JPF's virtual machine. BOX follows the second alternative. BOX provides a library for storing, partially restoring state, and re-executing the program. The user needs to write a test driver and instrument the program specially to allow state restoration (see Section 4.3.2 for details). Verisoft [God97] and CMC [MPC$^+$02] follow the third design. The model checker runs directly on top of the host machine.

We want to investigate how the alternatives above impact efficiency in model checkers that operate on special state representations and on top of a regular JVM, such as JPF.

Chapter 4

Delta Execution

This chapter presents Delta Execution, referred to as Δ*Execution,* a technique that can reduce overall exploration time in explicit-state model checkers.

ΔExecution performs program execution *simultaneously on several states.* The technique exploits the fact that many execution paths in state-space exploration partially overlap. ΔExecution speeds up the state-space exploration by sharing the common parts across the executions and separately executing only the "deltas" where the executions differ. Central to ΔExecution is the use of an *efficient representation and manipulation of sets of states.*

Symbolic model checking (SMC) [JEK$^+$90, CGP99] inspired ΔExecution. SMC enabled a breakthrough in model checking as it provided a much more efficient exploration than explicit-state model checking. Conceptually, SMC executes the program on a set of states and exploits the similarity among executions. Typical implementations of SMC represent states with Binary Decision Diagram (BDD) [Bry92], a data structure that supports efficient operations on boolean functions. However, heap operations prevent the direct use of BDDs for object-oriented programs. Although heaps are easily translated into boolean functions [LH04, WL04], the heap operations – including field reads and writes, dynamic object allocation, garbage collection, and comparisons based on heap symmetry [CGP99, LV01, Ios01, BKM02, MD05] – do not translate directly into efficient BDD operations. ΔExecution considers states that include heap. We focus our attention to model checking of programs that use dynamically allocated data, such as object-oriented programs.

ΔExecution executes the programs on set of states. As we will show, this special form of execution can be less expensive due to the presence of constants which are valid across all states in the set. In addition, it can also optimize path exploration and path pruning, which are two important and costly operations in explicit-state model checking.

We implemented ΔExecution in two model checkers: JPF and BOX (from *Bounded Object*

eXploration). As presented in Section 2.6, JPF is a popular general-purpose model checker for Java programs. BOX, in contrast, is a specialized model checker that we developed for efficient exploration of sequential Java programs. These implementations enabled the evaluation of ΔExecution on model checkers that follow different design principles. For instance, we identified that ΔExecution improves the overall exploration time in both model checkers, but the improvement is due to different reasons.

We used two scenarios of exploration in our evaluation. The first scenario applies ΔExecution using exhaustive exploration (See Section 1.5.1). The second applies ΔExecution in conjunction with *abstract matching*, a non-exhaustive technique for state space exploration that Visser et al. [VPP06] recently proposed. Similarly to mixed execution, ΔExecution becomes more effective with the increase in the number of states the model checker explores. We control this variable with the size of the method sequence and range of input used in a driver that guides the state space exploration (See Section 1.5.1).

In the exhaustive exploration, we first evaluate ΔExecution on ten simple subject programs. The results show that the technique improves exploration time from 0.88x to 126.80x (with median 5.60x) in JPF and from 0.58x to 4.16x (with median 2.23x) in BOX, while taking from 0.46x to 11.50x (with median 1.48x) less memory in JPF and from 0.18x to 2.71x (with median 1.18x) memory in BOX. Then, we evaluate the technique using exhaustive exploration on the AODV [PR99] case-study. AODV refers to an implementation of a networking protocol with errors (See details in Section 2.7). ΔExecution improved exploration time for AODV from 0.88x to 2.04x (with median 1.72x).

Our results also show that ΔExecution improves exploration time from 0.92x to 6.28x (with median 4.52x) in JPF with abstract matching. We conducted this experiment with 4 of the 10 simple programs used with exhaustive exploration. The relative decrease of improvement in ΔExecution under this scenario is due to the fact that abstract matching reduces the total number of states in the state space.

Next we show an example that illustrates key aspects of the technique. Then we detail the technique, discuss its implementation using the JPF and BOX model checkers, and discuss the experimental results. Finally, we discuss limitations and future work.

4.1 Example

We next present an example that illustrates how ΔExecution speeds up the state-space exploration compared to *standard execution*, i.e., to the the regular form of execution.

Figure 1.1, appearing in Section 1.3, shows a binary search tree class that implements a set. Each `BST` object stores the size of the tree and its root node, and each `Node` object stores an integer value and references to the two children. The `BST` class has methods to add and remove tree elements. A test sequence for the binary search tree class consists of a sequence of method calls, for example `BST t = new BST(); t.add(1); t.remove(2)`.

4.1.1 The standard driver

Figure 1.4 shows an example driver program that enables a model checker to systematically explore different states of the tree, i.e., to generate sequences of method calls that reproduce different states of the tree. This driver operates using standard execution and is thus called the *standard driver*. Figure 1.5 illustrates the states the model checker visits when using this driver to perform the state space exploration. We omit from the graph edges that correspond to invocations of `delete` as each "delete" edge is either a self-loop or leads to some existing state. The model checker applies breadth-first search to avoid missing important parts of the state space (see Section 1.5.1 for details).

Similarity of paths

Figure 1.3, appearing in Section 1.4, shows at the top states of trees with size 3. The exploration executes `add(4)` on each of the five trees of size 3, i.e., the standard driver separately executes `add(4)` on each pre-state, resulting in the five post-states shown at the bottom of the figure.

We use the term *individual execution* to emphasize the use of a single state in standard execution. A sequence of program counters defines one *path*. As we focus on sequential programs we also relate a path to the sequence of branching decision that an execution makes, i.e., the sequence of decisions also defines a path. We therefore informally say that *execution follows a path*: execution can follow that sequence of branching decisions to reconstruct that path. Also, we informally say that a *state follows a path* to denote that the execution starting with that state follows a path.

Important to note on the example above is that the executions of `add(4)` on some of these states follow the same path, i.e., each individual execution makes the *same sequence of branching decisions*. For instance, for the balanced tree and the tree to its left, execution adds a new node with value 4 to the right of the right child of the root. In either case, execution follows the path given by the following sequence of program counters (see Section 1.1): 1, 2, 4, 5, 6, 7, 10, 5, 6, 7, 8, 9, 17. We will show that explicit-state model checking can leverage on this similarity with the use of ΔExecution. This example shows the particular case when two executions have identical paths. It is important to stress that ΔExecution exploits the commonalities among program paths; it does not require paths to be identical.

4.1.2 The delta driver

Figure 4.1 shows a driver that explores states using ΔExecution. We refer to this driver as the *delta driver*. The delta driver is similar to the standard driver: both use non-deterministic choices to select different methods and input values, both prune the exploration based on the state of `bst`, and both use breadth-first exploration. However, the delta driver differs from the standard driver in the way it operates on the state. First, in the delta driver, the variable `bst` represents several individual trees (i.e., states that correspond to a standard execution.). We use the term $\Delta State$ to refer to a form of state that includes several individual states. Second, the delta driver backtracks the state differently than the standard driver. Specifically, the method `newIteration` returns one ΔState containing all individual states that should be explored in a given iteration. In the first iteration, this ΔState is a singleton that has only the initial state (i.e., the empty tree). The method `merge` at the end of one method execution path "collects" those trees (from `bst`) that have not been previously visited and thus should be explored in the next loop iteration. Effectively, the `merge` method combines all distinct states reachable with the method sequences of length `i` into one ΔState that the iteration `i+1` will use. The method `newValue` updates the internal state for ΔExecution as backtracking should not restore some parts of that internal state (specifically the statemask discussed in Section 4.2.2).

```
// N bounds sequence length and parameter values
public static void mainDelta(int N) {
  BST bst = new BST(); // empty tree
  for (int i = 0; i < N; i++) {
    bst = Delta.newIteration(bst);
    int methNum = Verify.getInt(0, 1);
    int value = Verify.getInt(1, N);
    Delta.newValue();
    switch (methNum) {
      case 0: bst.add(value); break;
      case 1: bst.remove(value); break;
    }
    Delta.merge(bst);
  }
}
```

Figure 4.1: Driver for delta execution.

Split and Merge

While standard execution invokes `add(4)` separately against each standard state, ΔExecution invokes `add(4)` simultaneously against *a set of standard states*. ΔExecution itself operates on one state, the Δ*State* . We call the operation that combines standard states into a ΔState *merging*. The top of Figure 1.3 illustrates one set consisting of the five pre-states. (Section 4.2.1 describes how to efficiently represent a ΔState, and Section 4.2.5 describes how to efficiently merge states.)

During program execution, ΔExecution occasionally needs to *split* the ΔState. For `add(4)`, for example, the five pre-states follow the same execution path until the first check of `temp.right == null`. At that point, ΔExecution splits the set of states: one subset (of two states) follows the `true` branch, and the other subset (of three states) follows the `false` branch. Note that the split enforces the invariant that all states in a set follow the same path.

Each split introduces a non-deterministic choice point in the execution. For `add(4)`, one execution with two states terminates after creating a node with value 4 and assigning it to the right of the root. The figure depicts this execution with the left arrow. The other execution with three states splits at the second check of `temp.right == null`: two (middle) states follow the `true` branch, and one (rightmost) state follows the `false` branch. These two executions terminate without further splits, appropriately adding the value 4 to the final trees. Note that ΔExecution produces the same number of states as in standard execution but it may result in less executions.

We next describe *merging*; the operation that ΔExecution performs to build a ΔState from

individual states. Merging is a dual operation of splitting: while splitting partitions a set of states into subsets, merging combines several sets of states (or several individual states) into a larger set. ∆Execution can, in principle, perform merging on any sets of states at any program point. For example, ∆Execution could merge all three sets of states from Figure 1.3 when they reach `size++`. However, our current implementation of ∆Execution considers only the program points that are method boundaries: it merges the states only after all of them finish the execution path for one method, since that is also where state comparison is done.

Performance

We next discuss how the performance of ∆Execution and standard execution compare. In our running example, ∆Execution requires only three execution paths to reach all five post-states that `add(4)` creates for the five pre-states. Additionally, these three paths share some prefixes that can be thus executed only once. In contrast, standard execution requires five executions of `add(4)`, one execution for each pre-state, to reach the five post-states. Also, each of these five separate executions needs to be executed for the entire path.

The experimental results show that ∆Execution is faster than standard execution for a number of subject programs and values for the bound N from the drivers. For example, for the binary search tree example and $N = 10$, ∆Execution speeds up JPF 7.11x and our model checker BOX 1.67x, while using over 2x more memory in JPF and 3x more memory in BOX. (On average, ∆Execution uses roughly the same amount of memory as standard execution; `BST` is one of the subjects where ∆Execution shows the smallest speedup. We use `BST` for illustrations due to its simplicity.)

As Chapter 1 shows, explicit-state model checking of programs include three major operations: (i) execution, (ii) path exploration, and (iii) path pruning. Execution of transitions refers to the operation that performs a deterministic step in the subject program, path exploration refers to the operation that performs exploration of *all* program paths created with non-deterministic choices, and path pruning refers to the operation that performs pruning of some of these paths based on some notion of equivalence, e.g., isomorphism of states that the model checker visits during the exploration. We refer to path exploration just as backtracking (backtracking is part of path exploration) and path pruning just as state comparison. It is important to state that we focus on

model checkers that perform backtracking by storing and restoring state and perform path pruning by comparing states during the state space exploration. (For a more detailed description of these operations, refer to Section 2.5.)

We summarize next how ΔExecution can improve the time to perform each of these operations.

- ΔExecution may reduce *execution* time. This happens because it is sometime possible to use constants valid across all states in the set. For example, an operation that reads the field `size` in either the pre or post-state takes constant time, instead of time linear to the number of states, because all trees have the same size. Figure 4.2 shows two measurements of constants in the exploration of the state space of trees. The graph on the left shows the ratio of the constant field accesses over the total number of field accesses. It shows how often constants appear in an iteration, but does not measure the savings related to these accesses. The graph to the right shows the savings in field access that constant in ΔExecution provides. If ΔExecution accesses a constant field in a set with n states, we count $n - 1$ savings: had not the constant been detected, execution would need to iterate over n states. The ratio of savings relate the total savings due to field accesses in ΔExecution with the total number of field accesses in standard execution, per iteration. For both metrics, the higher the value the better: execution benefits when constants appear often but mainly when the number of field accesses (reads or writes) decreases compared to standard execution. Both graphs show the measurement of constants with the increase of the iteration number in the delta driver. They show results for both the binary search tree in Figure 1.1 and an implementation of a red-black tree, with $N = 10$ (the maximum length of the method sequence the user provides to the standard and delta drivers.). Note that the effects of constants are more positive in trees which are "more balanced", as there are more trees in the set that share the same shape.

- The reduction in the number of executions enables less *backtracking*. Figure 1.3 illustrates that ΔExecution traverses only 3 of the 5 paths that lead each individual pre to a post-state. The paths only split when it is not possible for all the states in the current set to evaluate the branching condition to the same value.

- ΔExecution enables optimization of state *comparison*. This happens because it is possible
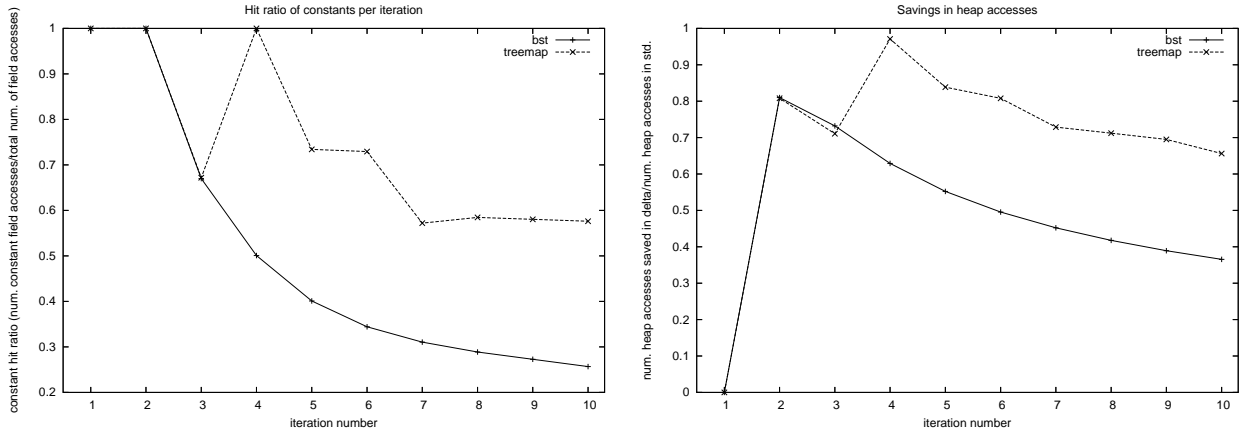
Figure 4.2: Metrics for constant field accesses. Plot to the left shows the hit ratio of field accesses in ΔExecution which are constant. The plot to the right shows the ratio of access savings in ΔExecution compared to the total number of accesses in standard execution.

to compute state linearizations (see Section 2.5) on set of states, i.e., to compute sets of linearizations simultaneously instead of one-by-one. In practice, this enables the linearization algorithm to internally share the prefixes of the linearization.

The trade-off between ΔExecution and standard execution can be summarized like this: ΔExecution performs fewer executions (avoiding separate execution of the same path shared by multiple states) than standard execution, but each execution in ΔExecution (that operates on a set of standard states) is more expensive than in standard execution (that operates on one standard state). It is also important to note that the presence of constants (i.e., values that are the same across a set of states) is essential to efficient operations under ΔExecution. Whether ΔExecution is faster or slower than standard execution for some exploration depends on several factors, including the number of execution paths, the number of splits, the cost to execute one path, the number of constants, and the sharing of execution prefixes.

4.2 Technique

The key idea of ΔExecution is to execute a program simultaneously on a set of standard states. We first discuss the representation the technique uses for set of individual states. We describe in detail two main operations on ΔStates: *splitting*, which divides a set of states into subsets for executing different program paths, and *merging*, which combines several states together into a set. We also

65
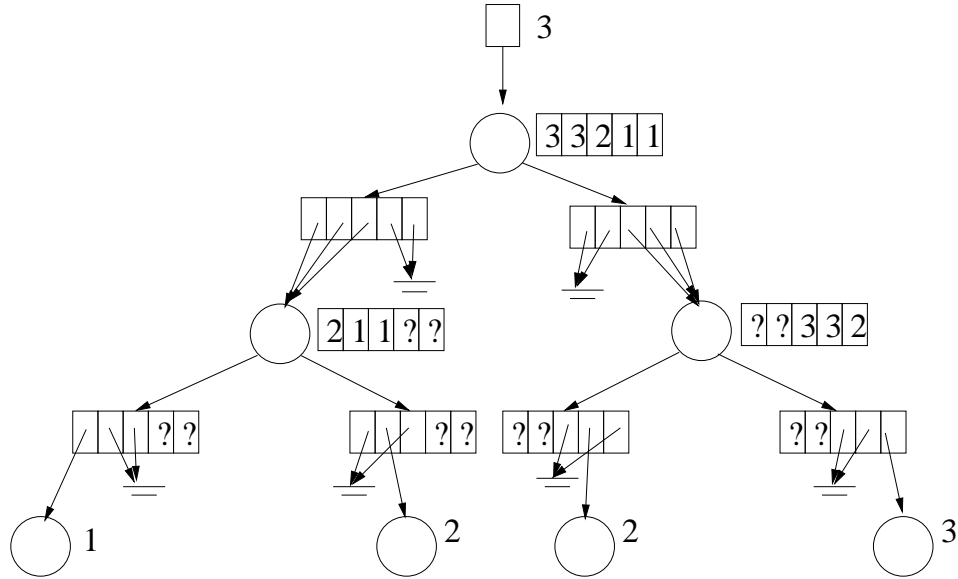
Figure 4.3: ΔState for the five pre-states from Figure 1.3.

present how program execution works in ΔExecution and how ΔExecution facilitates an optimized comparison of states.

4.2.1 ΔState

ΔExecution represents a set of individual standard states as a single Δ*State.* Each ΔState encodes all the information from the original individual states. A ΔState includes Δ*Objects* that can store multiple values (either references or primitives) that exist across the multiple individual states represented by a ΔState.

Figures 4.4, 4.5, and 4.6 show the classes used to represent ΔStates for the binary search tree example. We discuss here only the field declarations from those classes. (The methods from those classes implement the operations on ΔState and are explained later in the text.) Each object of the class `DeltaNode` stores a collection of references to `Node` objects, and each object of the class `DeltaInt` stores a collection of primitive integer values. The `BST` and `Node` objects are changed such that they have fields that are ΔObjects.

Figure 4.3 shows the ΔState that represents the set of five pre-states from Figure 1.3. Each ΔState consists of layers of "regular" objects and ΔObjects. In this ΔState, each of the pre-states has a corresponding *state index* that ranges from 0 to 4. Note that we could extract each of the

66

```
public class BST {
  private DeltaNode root = DeltaNode.NULL;
  private DeltaInt size = DeltaInt._new(0);

  public void add(DeltaInt info) {
    if (get_root().eq(DeltaNode.NULL))
      set_root(DeltaNode._new(info));
    else
      for (DeltaNode temp = get_root(); true; )
        if (temp.get_info().lt(info)) {
          if (temp.get_right().eq(DeltaNode.NULL)) {
            temp.set_right(DeltaNode._new(info));
            break;
          } else temp = temp.get_right();
        } else if (temp.get_info().gt(info)) {
          if (temp.get_left().eq(DeltaNode.NULL) {
            temp.set_left(DeltaNode._new(info));
            break;
          } else temp = temp.get_left();
        } else return; // no duplicates
    }
    set_size(get_size().add(DeltaInt._new(1)));
  }

  public DeltaBoolean remove(DeltaInt info) { ... }
}

class Node {
  DeltaNode left, right;
  DeltaInt info;
  Node(DeltaInt info) { this.info = info; }
}
```

Figure 4.4: Instrumented `BST` and `Node` classes.

five pre-states by traversing the ΔState while indexing it with the appropriate state index. For example, we can extract the balanced tree using state index 2. Also note that some of the values in the example ΔState are "don't cares" (labeled with "?") because the corresponding object is not reachable for that state index. For example, the first node to the left of the root has "?" in the field `info` for the last two states (with indexes 3 and 4) because those states have the value `null` for the field `root.left`.

While each ΔObject conceptually represents a collection of values, the implementation does not always need to use collections or arrays. In particular, a value is often constant across all (relevant) states. For example, the `info` fields for all tree leaves in Figure 4.3 have constant values (for the relevant states). Our implementation uses an *optimized representation for constants*. The optimization is straightforward, and we do not discuss it in detail. We point out, however, that the

67

```
class DeltaNode {
  // maps each state index to a Node object
  Node[] values; // conceptually

  DeltaNode(int size) { values = new Node[size]; }
  private DeltaNode(Node n) { values = new Node[]{ n }; }
  public static DeltaNode _new(DeltaInt info) {
    return new DeltaNode(new Node(info));
  }
  public boolean eq(DeltaNode arg) {
    StateMask sm = StateMask.getStateMask();
    StateMask trueMask = new StateMask(sm.size());
    StateMask falseMask = new StateMask(sm.size());
    foreach (int index : sm) {
      if (values[index] == arg.values[index]) { trueMask.enable(index); }
      else { falseMask.enable(index); }
    }
    boolean result;
    if (trueMask.isEmpty()) result = false;
    else if (falseMask.isEmpty()) result = true;
    else result = (Verify.getInt(0, 1) == 0); // split
    StateMask.setStateMask(result ? trueMask : falseMask);
    return result;
  }
  public DeltaNode get_left() {
    StateMask sm = StateMask.getStateMask();
    DeltaNode result = new DeltaNode(sm.size());
    foreach (int index : sm) {
      DeltaNode dn = values[index].left;
      result.values[index] = dn.values[index];
    }
    return result;
  }
  public void set_left(DeltaNode arg) {
    StateMask sm = StateMask.getStateMask();
    IdentitySet<Node> set = new IdentitySet<Node>();
    foreach (int index : sm) {
      Node n = values[index];
      if (set.add(n)) { n.left = n.left.clone(); /* true if n was added */ }
      n.left.values[index] = arg.values[index];
    }
  }
  public DeltaNode get_right() { ... }
  public void set_right(DeltaNode arg) { ... }
  public DeltaInt get_info() { ... }
  public void set_info(DeltaInt arg) { ... }
}
```

Figure 4.5: New `DeltaNode` class.

optimization is important both for reducing the memory requirements of ΔStates and for improving

the efficiency of operations on ΔStates.

68

```
class DeltaInt {
  // maps each state index to an integer value
  int[] values; // conceptually

  DeltaInt add(DeltaInt arg) {
    StateMask sm = StateMask.getStateMask();
    DeltaInt result = new DeltaInt(sm.size());
    foreach (int index : sm)
      result.values[index] = values[index] + arg.values[index];
    return result;
  }
  ...
}
```

Figure 4.6: Part of `DeltaInt` library class.

4.2.2 Splitting

ΔExecution operates on a ΔState that represents a set of standard states. ΔExecution can perform many operations on the entire set. It needs to *split* the set only at a branch control point (e.g., an `if` statement) where some states from the set evaluate to different branch outcomes (e.g., for one subset of states, the branch condition evaluates to true, and for the other subset of states, it evaluates to false). We call such points *split points*; effectively, they introduce non-deterministic choice points as ΔExecution needs to explore both outcomes. (Note that no split is necessary even for branch control points when all states evaluate to the same branch outcome.)

One challenge in ΔExecution is to efficiently split ΔStates. Our solution is to introduce a *statemask* that identifies the currently *active states* within a ΔState. Each statemask is a set of state indexes. At the beginning of an execution, ΔExecution initializes the statemask to the set of all state indexes. For example, the execution of `add(4)` for the ΔState from Figure 4.3 starts with the statemask being $\{0, 1, 2, 3, 4\}$.

At the appropriate branch points, ΔExecution needs to split the set of states into two subsets. Our approach does not explicitly divide a ΔState into two ΔStates; instead, it simply changes the statemaskto reflect the splitting of the set of states. Specifically, ΔExecution builds a new statemask to identify the new subset of active states in the ΔState. It also saves the statemask for the other subset that should be explored later on. The execution then proceeds with the new subset.

After ΔExecution finishes the execution path for some (sub)set of states, it *backtracks* to some

unexplored split point to explore the other path using the statemask saved at the split point. Backtracking changes the statemask but restores the ΔState to exactly what it was at the split point. A model checker can implement backtracking in several ways. JPF, for instance, stores and restores state while BOX uses program re-execution. Section 4.3 elaborates this discussion.

To illustrate how the statemaskchanges during the execution, consider the example from Figure 1.3. The statemask is initially $\{0, 1, 2, 3, 4\}$. At the first split point, the execution proceeds with the statemask being $\{0, 1\}$. After the first backtracking, the statemask is set to $\{2, 3, 4\}$. At the second split point, the execution proceeds with the statemask being $\{2, 3\}$. After the second backtracking, the statemask is set to $\{4\}$ for the final execution.

Appropriate use of a statemask can facilitate optimizations on the ΔState. Consider, for example, a ΔObject that is not a constant when all states are active. This object can temporarily be transformed into a constant if all its values are the same for some statemask occurring during the execution. For instance, in our running example, the value of `root.right` becomes the constant `null` when the statemask is $\{0, 1\}$. Additionally, the statemask allows the use of *sparse representations* for ΔObjects: instead of using an array to map all possible state indexes into values, a sparse ΔObject can use representations that *map only the active state indexes into values*, thereby reducing the memory requirement.

4.2.3   Program execution model

We next discuss how ΔExecution executes program operations. The key is to execute each operation simultaneously on a set of values. ΔExecution uses a non-standard program execution that manipulates a ΔState that represents a set of standard states. Such non-standard execution can be implemented in two ways: (i) instrumenting the code such that the regular execution of the instrumented code corresponds to the non-standard execution [KPV03, VPK04, XMSN05] or (ii) changing the execution engine such that it interprets the operations in the non-standard semantics [dPX$^+$06]. Our current implementation uses instrumentation: the subject code is pre-processed to support ΔExecution.

We use parts of the instrumentation to describe the semantics of ΔExecution.

Classes

The instrumentation changes the original program classes and generates new classes for ΔObjects. Figure 1.1 from Chapter 1 shows a part of the original code for the binary search tree example. Figures 4.4, 4.5, and 4.6 show the key parts of the instrumented code for this example. Figure 4.4 shows the instrumented version of the original `BST` and `Node` classes. Figure 4.5 shows the new class `DeltaNode` that stores and manipulates the multiple `Node` references that can exist across the multiple states in a ΔState. Figure 4.6 shows the class `DeltaInt` that stores and manipulates multiple `int` values; this class is a part of the ΔExecution library and is not generated anew for each program.

It is important to note that ΔObjects are immutable from the perspective of the instrumented code in the same way that regular primitive and reference values are immutable for standard execution. This allows sharing of ΔObjects. For example, this allows direct assignment of one `DeltaInt` object to another (e.g., `int x = y` simply becomes `DeltaInt x = y`). Our implementation internally mutates ΔObjects to achieve higher performance, in particular when values become constant across active states. The mutation handles the situations that involve shared ΔObjects and require a "copy-on-write" cloning.

Types

The instrumentation changes all types in the original program to their delta versions. Comparing figures 1.1 and 4.4, notice that the occurrences of `Node` and `int` have been replaced with the new `DeltaNode` class (from Figure 4.5) and the `DeltaInt` class (from Figure 4.6), respectively. The instrumentation also appropriately changes all definitions and uses of fields, variables, and method parameters to use ΔObjects.

Field accesses

The instrumentation replaces standard object field reads and writes with calls to new methods that read and write fields across multiple objects. For example, all reads and writes of `Node` fields are replaced with calls to getter and setter methods in `DeltaNode`. Consider, for instance, the field read `temp.left`. In ΔExecution, `temp` is no longer a reference to a single `Node` object but a reference

71

to a `DeltaNode` object that tracks multiple references to possibly many different `Node` objects. The `left` field of `Node` is now accessed via the `get_left` method in `DeltaNode`. This method returns a `DeltaNode` object that references (one or more) `Node` objects that correspond to the `left` fields of all `temp` objects whose states are active in the statemask. In general, this can result in an execution split when some objects in `temp` are `null`.

Operations

The instrumentation replaces (relational and arithmetic) operations on reference and primitive values with method calls to `DeltaNode` and `DeltaInt` objects. All original operations on values now operate on ΔObjects that represent sets of values. More precisely, the methods in ΔObjects do not need to operate on all values but only on those values that correspond to the active state indexes as indicated by the statemask.

Consider integer addition as an example of arithmetic operation. In standard execution, addition takes two integer values and creates a single value. In ΔExecution, it takes two `DeltaInt` objects and creates a new `DeltaInt` object. The `add` method in `DeltaInt` (from Figure 4.6) shows how ΔExecution conceptually performs pairwise addition across all active state indexes for the two `DeltaInt` objects. Our implementation optimizes the cases when those objects are constant (to avoid the loop or state indexing).

For an example relational operation, consider reference equality. The method `eq` in `DeltaNode` (from Figure 4.5) performs this operation across all active state indexes. Note that this method can create a split point in the execution if the result of the operation differs across the states. If so, `eq` introduces a non-deterministic choice (with `getInt`) that returns a boolean `true` or `false` after appropriately setting the statemask.

Method calls

The instrumentation replaces a standard method call with a method call whose receiver is a ΔObject, which allows making the call on several objects at once. Note that each call introduces a semantic branch point (since different objects may have different dynamic types) and can result in an execution split.

72

4.2.4 Optimized state comparison

Heap symmetry [MD05, LV01, CGP99, Ios01] is an important technique that model checkers use to alleviate the state-space explosion problem. Heap symmetry detects equivalent states: when the exploration encounters a state equivalent to some already visited, the exploration path can be pruned. In object-oriented programs, two heaps are equivalent if they are *isomorphic* (i.e., have the same structure and primitive values, while their object identities can vary) [Ios01, MD05, BKM02]. An efficient way to compare states for isomorphism is to use *linearization* (also known as serialization or marshalling) that translates a heap into a sequence of integers such that two heaps are isomorphic if and only if their linearizations are equal.

ΔExecution exploits the fact that different heaps in a ΔState can share prefixes of linearization. Instead of computing linearizations separately for each state in a set of states, ΔExecution *simultaneously computes a set of linearizations* for a ΔState. Sharing the computation for the prefixes not only reduces the execution time but also reduces memory requirements as it enables sharing among the sequences used for linearizations.

We next present how to transform a basic algorithm that separately linearizes each state from a ΔState into an efficient algorithm that simultaneously linearizes all states from a ΔState. Figure 4.7 shows a pseudo-code of a basic algorithm that iterates over each active state from the statemaskand computes the linearization for the individual state. For simplicity of presentation, this algorithm assumes that the heaps contain only reference fields of only one class. Our actual implementation handles general heaps with objects of different classes, primitive fields, and arrays.

The method `linObject` produces a sequence of integers that represent linearization for the state reachable from `o`. When `o` is `null`, `linObject` returns a singleton sequence with the value that represents `null`. When `o` is a reference to a previously linearized object, `linObject` returns a singleton sequence with the identifier used for that object, which handles object aliasing. The map `ids` stores the association between objects and their ids. When `o` is an object not yet linearized, `linObject` creates a new id for it, appropriately extends the map, and linearizes all the object fields.

The method `linFields` linearizes the fields of a given object. A typical implementation is iterative, as shown in the first `linFields` method. It is important to note that the value of the expression `o.getField(f).values[index]` determines the linearizations for different states. We target this ex-

73

```
void linearize(Object o, smask sm) {
  foreach (int index : sm) {
    Pair(Map _, Seq s) = linObject(o, new Map(), index);
    checkVisited(index, s);
  }
}


Pair<Map, Seq> linObject(Object o, Map ids, int index) {
  if (o == null) return Pair(ids, Seq(NULL));
  if (o in ids) return Pair(ids, Seq(ids.get(o)));
  int id = ids.size();
  return linFields(o, ids.put(o, id), Seq(id), index);
  /*return linFields(o, 0, ids.put(o, id), Seq(id), index);*/
}


Pair<Map, Seq> linFields(Object o, Map ids,
                         Seq seq, int index) {
  for (int f = 0; f < o.numberOfFields(); f++) {
    Object fo = o.getField(f).values[index];
    Pair(ids, Seq s) = linObject(fo, ids, index);
    seq = seq.append(s);
  }
  return Pair(ids, seq);
}


Pair<Map, Seq> linFields(Object o, int f, Map ids,
                         Seq seq, int index) {
  if (f < o.numberOfFields()) {
    Object fo = o.getField(f).values[index];
    Pair(Map m, Seq s) = linObject(fo, ids, index);
    return linFields(o, f + 1, m, seq.append(s), index);
  } else return Pair(ids, seq);
}
```

Figure 4.7: Non-optimized linearization of ∆State.

pression to be the split point in our optimized linearization algorithm. The algorithm thus needs to explore different execution paths from this point, effectively performing backtracking.

We want to implement the optimized algorithm, supporting explicit backtracking, on the regular JVM. An intermediate step in the optimization is to transform the algorithm to conceptually use the continuation-passing style [FWH01]. In practice, the method linFields is transformed into a recursive implementation shown in the second linFields method. This version exposes the field index f and linearizes the fields of o between f and o.getNumberOfFields(). This version permits the linearization to *continue* an execution from the point it was left at in linFields. Note that linFields and linObject manipulate functional objects Map and Seq, which facilitates backtracking of the state.

```
Stack stack; // mutable structure
void linearize(Object o, smask sm) {
  stack = new Stack();
  Triple(Map _, Seq s, smask tm) =
    linObject(o, new Map(), sm);
  checkVisited(tm, s); // all states from tm have sequence s
  while (!stack.isEmpty()) {
    Tuple(Object o, int f, Map ids,
          Seq seq, smask nm) = stack.pop();
    Triple(Map _, Seq s, smask tm) =
      linFields(o, f, ids, seq, nm);
    checkVisited(tm, s);
  }
}

Triple<Map, Seq, smask>
  linObject(Object o, Map ids, smask sm) {
  if (o == null) return Triple(ids, Seq(NULL), sm);
  if (o in ids) return Triple(ids, Seq(ids.get(o)), sm);
  int id = ids.size();
  return linFields(o, 0, ids.put(o, id), Seq(id), sm);
}

Triple<Map, Seq, smask>
  linFields(Object o, int f,
            Map ids, Seq seq, smask sm) {
  if (f < o.numberOfFields()) {
    Triple(Object fo, smask em, smask nm) =
      split(o.getField(f), sm);
    if (nm is not empty)
      stack.push(o, f, ids, seq, nm);
    Triple(smask om, Map m, Seq s) = linObject(fo, ids, em);
    return linFields(o, f + 1, m, seq.append(s), om);
  } else return Triple(sm, ids, seq);
}
```

Figure 4.8: Optimized linearization of ΔState.

Figure 4.8 shows the pseudo-code of the optimized algorithm that linearizes a ΔState in the ΔExecution mode. The new methods `linObject` and `linFields` do not take one state index but take a statemask with several active state indexes to linearize. These methods now return a statemask and one linearization for all the states in that statemask. The linearization can introduce non-deterministic choices to enforce the invariant that all states in the statemask have the same linearization prefix. When the linearization completes for some statemask, it needs to backtrack to explore the remaining statemasks.

The `stack` object stores the backtracking points. Each entry stores the state that needs to be restored to continue an execution from a split point: the root object, the field index, the map for

object identifiers, the current linearization sequence, and the statemask. While `stack` is mutable, the other structures are immutable, which makes it easy to restore the state. The `while` loop in `linearize` visits each pending backtracking point until it finishes computing all linearizations.

The only source of non-determinism in the linearization is the reading of fields across different states from the statemask. The method `split` takes as input a ΔObject `do = o.getField(f)` and a statemask`sm`. It returns a standard object `fo = do.values[idx]` for some `idx` from `sm`, a statemask`em` (which comes from "equals mask") of `index` values such that `do.values[index] == fo`, and a statemask`nm` (which comes from "non-equal mask") of `index` values such that `do.values[index] != fo`. At this point, `linFields` first pushes on the stack an entry with the backtracking information for `nm` and then continues the linearization of `fo` for the states in `em`.

4.2.5 Merging

The dual of splitting sets of states into subsets is *merging* several sets of states into a larger set. Recall the driver for ΔExecution from Figure 4.1. It merges all non-visited states from one iteration into a ΔState to be used at the start of the next iteration. Specifically, the `merge` method receives as the input a ΔState and (implicitly) a statemask. This method extracts the non-visited states from the ΔState and only stores their linearized representations. The method `newIteration` builds and returns a new ΔState from the stored linearized representations.

Our merging uses *delinearization* to construct a ΔState from the linearized representations of non-visited states. The standard delinearization is an inverse of linearization: given one linearized representation, delinearization builds one heap isomorphic to the heap that was originally linearized. The novelty of our merging is that it operates on a *set* of linearized representations simultaneously and, instead of building a set of standard heaps, it builds one ΔState that encodes all the heaps. It is interesting to point out that we often used in debugging our implementation the fact that linearization and delinearization are inverses; the composition of these functions gives the identity function: for any set of linearizations s, the linearization of the delinearization of s should equal s.

We highlight two important aspects of the merging algorithm. First, it identifies ΔObjects that should be constants (with respect to the reachability of the nodes), which results in a more efficient ΔState. Such constants can occur quite often; for instance, in our experiments (see Section 4.4),

76

the lowest percentage of the constant ΔObjects in the merged ΔStates is 33%. Second, the merging algorithm *greedily* shares the objects in the resulting ΔState: it attempts to share the same ΔObject among as many individual states as possible. For example, in Figure 4.3, the left node from the root is shared among three of the five states.

Figure 4.9 shows the pseudo-code of our merging algorithm. The input is a collection of linearizations, and the output is a root object for a ΔState. The algorithm maintains a collection of maps from object ids to actual objects (which handles aliasing) and a collection of offsets that track progress through the different linearizations (since they do not need to go in a "lockstep"). The method `createObject` constructs *one* object shared for all states in the given statemask and invokes `createDeltaObject` to construct each field of the object. Note that this sharing does not constitute aliasing in the standard semantics since: only one reference is visible for any given state. The method `createDeltaObject` examines the field values across all states in the statemask `sm`. For each state, it checks for three possible options for the field's object id: (i) it denotes the `null` reference, (ii) it denotes an alias, or (iii) it denotes a new object. For the first two options, the algorithm assigns the value to the delta object `d` as it performs the check. For the third option, it just records in the state mask object `cm` the index of the state during the check. If the smask `cm` is not empty after the check across all states, the algorithm recursively invokes (once) `createObject` to create an object that will be shared among the states in `cm`. Lastly, the algorithm checks if the delta object `d` is semantically a constant, i.e., if it contains the same value across all states denoted by `sm`. A special constant object is created in that case.

For states that have aliases between objects (unlike binary search tree), this greedy algorithm does not always produce a ΔState with the smallest number of nodes, and some alternative algorithms could produce smaller graphs. Figure 4.10 illustrates an example where an alternative merging algorithm could find more sharing. Given the two states at the top, our greedy algorithm produces the ΔState at the bottom-left that does not share the two subgraphs denoted by shaded triangles. In contrast, a more complex algorithm could potentially identify this sharing opportunity and construct the ΔState at the bottom-right. However, such alternative algorithms would require more time to search for appropriate sharing opportunities that result in smaller ΔStates.

```
Seq[] lin; // input
Map<int, Object>[] maps; // intermediate result, mutable
int[] offsets; // intermediate result, mutable
Object merge() {
  maps = new Map[lin.length](); // all empty maps
  offsets = new int[lin.length]; // all zeroes
  // the state mask starts as the set {0..lin.length-1}
  return createObject(new StateMask(lin.length));
}
Object createObject(StateMask sm) {
  Object o = new Object();
  foreach (int index : sm) {
    int id = lin[index][offsets[index]++];
    maps[index].put(id, o);
  }
  foreach (field f in o) o.f = createDeltaObject(sm);
  return o;
}
DeltaObject createDeltaObject(StateMask sm) {
  DeltaObject d = new DeltaObject(lin.length);
  // state indexes for which to create a new object
  StateMask cm = new StateMask();
  foreach (int index : sm) {
    int id = lin[index][offsets[index]++];
    if (id == NULL) d.values[index] = null;
    else if (maps[index].contains(id))
      d.values[index] = maps[index].get(id);
    else { // need to create a new object for this id
      cm.add(index); offsets[index]--; }
  }
  if (cm not empty) {
    // key: greedily sharing the new object across indexes
    Object co = createObject(cm);
    foreach (int index : cm) d.values[index] = co;
  }
  // optimization for constants
  if (d.values is constant with respect to sm)
    d = new DeltaObjectConstant(d.values[some index from sm]);
  return d;
}
```

Figure 4.9: Pseudo-code of the merging algorithm.

4.3 Implementation

We implemented ΔExecution in two model checkers, JPF and BOX. JPF [VHB+03] is a popular model checker for Java programs, but it is general-purpose and has a high overhead [dSM06] for the subject programs considered in our study and related studies [VPP05,VPP06,dSM06]. For the purpose of evaluating the technique under different implementations, we also implemented BOX (*Bounded Object eXploration*), a model checker specialized for sequential programs.
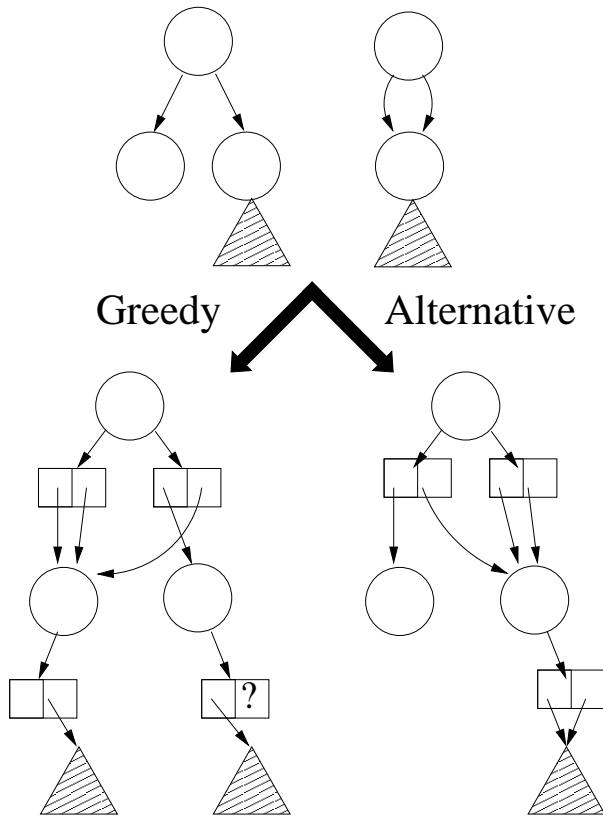
Figure 4.10: Greedy vs. alternative merging.

4.3.1 JPF

We implemented ΔExecution by modifying JPF version 4 [jpf]. JPF is implemented as a backtrackable Java Virtual Machine (JVM) running on top of a regular, host JVM. JPF provides operations for state-space exploration: storing states, restoring them during backtracking, and comparing them. By default, JPF compares the entire JVM state that consists of the heap, stack (for each thread), and class-info area (that is mostly static but can be modified due to the dynamic class loading in Java). However, our experiments require only the part of the heap reachable from the root object in the driver. We therefore disabled the JPF's default state comparison and instead use a specialized state comparison as done in some previous studies with JPF [XMSN05,VPP06,dPX$^+$06].

We next discuss how we implemented each component of ΔExecution in JPF. We call the resulting system ΔJPF. ΔJPF keeps ΔState as a part of the JPF state, which enables the use of JPF backtracking to restore ΔState at the split points. We implemented the library operations on ΔState (such as arithmetic and relational operations or field reads and writes) to execute on the

host JVM. Effectively, the library forms an extension of JPF; our goal is not to model check the library itself but the subject code that uses the library. ΔJPF uses instrumented code to invoke the operations that manipulate the ΔState.

We implemented splitting in ΔJPF on top of the existing non-deterministic choices in JPF. It is important to point out that our implementation leverages JPF to restore the entire ΔState but uses statemasks to indicate the active states. Therefore, ΔJPF manages statemasks on the host JVM, outside of the backtracked state. We implemented merging also to execute on the host JVM and to create one ΔState as a JPF state that encodes all the non-visited states encountered in the previous iteration of the exploration. Recall that the drivers in our experiments use breadth-first exploration.

To automate the instrumentation of code for execution on ΔJPF, we developed a plug-in for Eclipse version 3.2 [Ecl]. This plug-in takes a subject program and manipulates its Eclipse internal AST representation to automate the steps described in Section 4.2.3.

4.3.2 BOX

We developed BOX, a model checker optimized for sequential Java programs. JPF is a general-purpose model checker for Java that can handle concurrent code and can store, restore, and compare the entire JVM state that consists of heap, stack, and class-info area. However, in unit testing of object-oriented programs, most code is sequential and most drivers need to store, restore, and compare only the heap part of the state. Therefore, we used the existing ideas from state-space exploration research [God97, VHB$^+$03, Ios01, Fou, CDH$^+$00, RDH03, MPC$^+$02, AQR$^+$04] to engineer a high-performance model checker for such cases.

BOX can store/restore/compare only a part of the program heap reachable from a given root. The root corresponds to the main object under exploration in the driver. BOX uses a *stateful* exploration (by restoring the entire state) *across iterations* and *stateless* exploration (by re-executing one method at a time) *within one iteration*. BOX needs to re-execute a method within an iteration as it does not store the state of the program stack. Instead, BOX only keeps a list of changes performed on the heap during a single method execution and restores the state by undoing those changes. For efficient manipulation of the changes, BOX requires that code under exploration be

instrumented.

We refer to the ΔExecution implementation in BOX as ΔBOX. ΔBOX needs to backtrack the ΔState in order to explore a method for various statemasks. ΔBOX *re-executes* the method from the beginning to reach the latest split point. While re-execution is seemingly slow, it can actually work extremely well in many situations. For example, Verisoft [God97] is a well-known model checker that effectively employs re-execution.

ΔBOX implements the components of ΔExecution as presented in Section 4.2. ΔBOX represents ΔState as a regular Java state that contains both ΔObjects and objects of the instrumented classes. Our instrumentation for ΔBOX (as well as for BOX) is mostly manual at this time. ΔBOX uses instrumented code to perform the operations on the ΔState. Similarly to ΔJPF, ΔBOX merges states between iterations of the breadth-first exploration. ΔBOX always employs the optimized state comparison as presented in Section 4.2.4.

4.4 Evaluation

We present an experimental evaluation of ΔExecution. We first describe the ten basic subject programs used in the evaluation and then discuss the improvements that ΔExecution provides for an exhaustive exploration of these programs in both JPF and BOX and a non-exhaustive exploration in JPF. We finally present the improvements that ΔExecution provides on a larger case study, an implementation of the AODV routing protocol [PR99].

We performed all experiments on a Pentium 4 3.4GHz workstation running under RedHat Enterprise Linux 4. We used Sun's JVM 1.5.0_07, limiting each run to 1.8GB of memory and 1 hour of elapsed time.

4.4.1 Basic subjects

We evaluated ΔExecution on ten subject programs taken from a variety of sources. All but one of these subjects have been previously used to evaluate testing and model-checking techniques. The following nine subjects are data structures:

- `binheap` is an implementation of priority queues using binomial heaps [VPP06]

- `bst` is our running example that implements a set using binary search trees [BKM02,XMSN05]

- `deque` is our implementation of a double-ended queue using doubly-linked lists

- `fibheap` is an implementation of priority queues using Fibonacci heaps [VPP06]

- `heaparray` is an array-based implementation of priority queues [BKM02, XMSN05]

- `queue` is an object queue implemented using two stacks [DB06]

- `stack` is an object stack [DB06]

- `treemap` is an implementation of maps using red-black trees based on Java collection 1.4 [BKM02, XMSN05, VPP06]

- `ubstack` is an array-based implementation of a stack bounded in size, storing integers without repetition [SLA02, XMN04, CS04, PE05]

The tenth subject is `filesystem`, which is based on the Daisy file-system code [Qad]. While the original code had seeded errors, we use a corrected version provided by Darga and Boyapati and used in another study [DB06]. The primary purpose of our evaluation is to compare the efficiency of ΔExecution and standard execution, so we use correct implementations of all basic subjects. (The AODV case study described in Section 4.4.4 uses code with errors that violate a safety property.)

For each subject, we wrote drivers for standard execution and for ΔExecution (similar to figures 1.4 and 4.1). The drivers exercise the main mutator methods. For data structures, the drivers add and remove elements. For `filesystem`, the drivers create and remove directories, create and remove files, and write to and read from files.

4.4.2 Exhaustive exploration

Table 4.1 shows the experimental results for exhaustive exploration. For each subject and several bounds (on the sequence length and parameter size, as in the driver shown in Figure 4.1), we tabulate the overall exploration time and peak memory use with and without ΔExecution in both JPF and BOX, and the characteristics of the explored state spaces. The cells marked with "*" indicates that the experiment either ran out of 1.8GB memory or exceeded the 1 hour time limit.

experiment		JPF time			JPF mem.	# states	# executions		
subject	N	std	delta	std/delta	std/delta		std	delta	std/delta
binheap	7	24.87	2.30	10.82x	1.16x	16864	236096	401	588
	8	458.81	11.92	38.50x	1.03x	250083	4001328	863	4636
	9	*	*	*	*	1353196	24357528	1069	22785
bst	9	44.02	7.86	5.60x	0.70x	46960	845280	10846	77
	10	214.06	30.13	7.11x	0.46x	206395	4127900	22688	181
	11	*	*	*	*	915641	20144102	46731	431
deque	8	54.70	4.13	13.25x	1.50x	69281	1108496	576	1924
	9	552.11	28.84	19.14x	1.48x	623530	11223540	810	13856
	10	*	*	*	*	6235301	124706020	1100	113369
fibheap	6	3.18	1.46	2.17x	0.98x	3003	21021	82	256
	7	25.09	2.82	8.90x	2.13x	36730	293840	130	2260
	8	400.84	21.59	18.57x	0.88x	544659	4901931	209	23454
filesystem	3	1.98	1.88	1.06x	0.97x	58	6264	576	10
	4	17.18	3.08	5.59x	11.50x	1353	194832	1568	124
	5	*	*	*	*	64576	11623680	3940	2950
heaparray	8	104.96	3.61	29.09x	2.31x	97092	873828	258	3386
	9	2,724.63	21.49	126.80x	1.22x	804809	8048090	359	22418
	10	*	*	*	*	8722946	95952406	488	196623
queue	6	6.46	1.46	4.42x	2.64x	10057	70399	45	1564
	7	84.42	5.08	16.63x	1.77x	147995	1183960	60	19732
	8	*	*	*	*	2578641	23207769	77	301399
stack	6	5.00	1.41	3.55x	1.01x	9331	65317	42	1555
	7	59.70	4.14	14.43x	1.31x	137257	1098056	56	19608
	8	*	*	*	*	2396745	21570705	72	299593
treemap	12	274.26	53.40	5.14x	3.44x	96401	2313624	7774	297
	13	871.16	160.75	5.42x	3.90x	282532	7345832	11105	661
	14	2,860.23	562.70	5.08x	4.41x	844655	23650340	15178	1558
ubstack	8	61.52	4.60	13.37x	1.57x	109681	987129	595	1659
	9	1,502.24	32.54	46.17x	1.48x	991189	9911890	931	10646
	10	*	*	*	*	9922641	109149051	1414	77191

Table 4.1: Overall time and memory for exhaustive exploration, using ΔExecution in JPF, and characteristics of the explored state spaces. We use the symbol * to indicate that an experiment either run out of memory or time (1h). Second is the unit of time.

The columns labeled "std/delta" show the improvements that ΔExecution provides over standard execution. Note that the numbers are ratios and not percentages; for example, for `binheap` and $N = 7$, the ratio is 10.82x, which corresponds to about 90% improvement $(= (1 - 1/ratio) * 100)$. For JPF, the speedup ranges from 0.88x (for `aodv` and $N = 6$) to 126.80x (for `aodv` and $N = 10$), with median 5.60x. For BOX, the speedup ranges from 0.58x (for `fibheap` and $N = 8$, which actually represents almost a 2x slowdown) to 4.16x (for `fibheap` and $N = 8$), with median 2.23x.

Note that the ratio less than 1x means that ΔExecution ran slower (or required more memory) than standard execution, for example for `filesystem` and $N = 3$ in BOX. While this can happen for smaller bounds, ΔExecution consistently runs faster than standard execution for important cases with larger bounds.

experiment		BOX time			BOX mem.
subject	N	std	delta	std/delta	std/delta
binheap	7	0.78	0.35	2.23x	2.71x
	8	11.63	3.38	3.44x	1.08x
	9	106.54	32.74	3.25x	1.04x
bst	9	2.42	1.53	1.59x	0.77x
	10	12.55	7.51	1.67x	0.30x
	11	67.64	49.62	1.36x	0.18x
deque	8	2.20	0.77	2.86x	1.54x
	9	22.38	7.48	2.99x	1.14x
	10	281.84	99.77	2.82x	1.18x
fibheap	6	0.22	0.16	1.40x	-
	7	1.16	0.66	1.76x	1.24x
	8	16.77	9.75	1.72x	0.68x
filesystem	3	0.14	0.25	0.58x	-
	4	1.18	0.71	1.67x	1.72x
	5	37.43	30.04	1.25x	0.97x
heaparray	8	1.21	0.88	1.37x	1.24x
	9	11.92	8.91	1.34x	0.53x
	10	127.10	110.26	1.15x	0.58x
queue	6	0.37	0.16	2.25x	-
	7	3.87	0.93	4.16x	1.44x
	8	78.62	25.36	3.10x	1.00x
stack	6	0.31	0.12	2.55x	-
	7	2.92	0.71	4.09x	1.87x
	8	59.98	17.81	3.37x	1.31x
treemap	12	32.88	9.12	3.61x	1.34x
	13	102.85	29.02	3.54x	1.48x
	14	365.54	104.09	3.51x	2.48x
ubstack	8	2.26	1.28	1.77x	1.30x
	9	22.60	13.52	1.67x	0.66x
	10	265.49	174.96	1.52x	0.62x

Table 4.2: Overall time and memory for exhaustive exploration in BOX. The characteristics of the state space appears in Table 4.1. We use the symbol - to indicate that the measurement of memory is not reliable due to the short run. Second is the unit of time.

ΔExecution provides these significant improvements because it exploits the overlap among executions in the state-space exploration. Table 4.1 shows the information about the state spaces explored in the experiments. Note that the number of explored states is the same with and without ΔExecution. This is as expected: ΔExecution focuses on improving the exploration time and does not change the exploration itself. (We used the difference in the number of states to debug our implementations of ΔExecution.) However, the numbers of executions with and without ΔExecution do differ, and the column labeled "std/delta" under the group "#executions" shows the ratio of the numbers of executions. The ratio ranges from 10x to 301399x. While this ratio effectively

experiment		standard execution time			ΔExecution time			
subject	N	exec.	comp.	backt.	exec.	comp.	backt.	merg.
binheap	7	17.62	0.54	6.71	0.59	0.26	1.12	0.34
	8	364.45	4.90	89.46	3.99	2.21	1.24	4.48
bst	9	20.44	4.20	19.39	2.25	2.40	1.94	1.27
	10	103.39	21.04	89.62	7.18	12.85	3.98	6.12
deque	8	25.50	3.45	25.75	0.72	1.08	1.18	1.14
	9	267.42	38.31	246.38	6.37	12.19	1.26	9.02
fibheap	6	1.25	0.11	1.81	0.18	0.08	1.08	0.13
	7	14.69	0.86	9.53	0.42	0.31	1.20	0.89
	8	256.79	8.02	136.03	4.07	4.49	1.41	11.63
filesystem	3	0.24	0.05	1.69	0.20	0.15	1.46	0.07
	4	4.67	0.46	12.04	0.60	0.69	1.59	0.20
heaparray	8	15.10	1.72	88.13	1.10	0.38	1.13	1.00
	9	160.36	17.38	2,546.90	8.85	4.40	1.36	6.87
queue	6	3.07	0.15	3.24	0.04	0.07	1.11	0.24
	7	48.30	1.52	34.60	0.18	0.70	1.10	3.09
stack	6	1.77	0.10	3.13	0.02	0.06	1.13	0.20
	7	28.38	1.85	29.46	0.02	0.49	1.18	2.44
treemap	12	191.51	26.06	56.70	5.05	43.52	2.00	2.83
	13	622.58	81.12	167.46	13.11	137.08	2.10	8.46
	14	2,031.64	283.39	545.19	38.45	494.99	2.47	26.78
ubstack	8	31.95	2.58	26.99	1.34	0.97	1.13	1.15
	9	357.06	30.19	1,114.99	14.09	9.06	1.37	8.02

Table 4.3: Time breakdown for JPF experiments.

enables ΔExecution to provide the speedup, there is no strict correlation between the ratio and the speedup. The overall exploration time depends on several factors, including the number of execution paths, the number of splits, the cost to execute one path, the frequency of constants in ΔStates, and the sharing of execution prefixes.

Time

We next discuss in more detail where state-space exploration spends time. In particular, where ΔExecution reduces time. Each state-space exploration, standard and Δ, includes three components: (i) execution, (ii) path exploration, and (iii) path pruning. ΔExecution additionally includes (iv) merging. As we discuss in Section 2.5, we use the terms backtracking and state comparison to refer to path exploration and path pruning, respectively. Table 4.1 and 4.4 show the breakdown of the overall exploration time on these four components for JPF and BOX. We show the numbers for only some of the experiments (for subjects from Section 4.4.3); the conclusions are the same for the other experiments.

In JPF, ΔExecution significantly reduces the time for code execution and state backtracking. For example, for `binheap` and $N = 7$, ΔExecution reduces the execution time from 17.62s to 0.59s and the backtracking time from 6.71s to 1.12s. These savings are big enough and make the times for merging and state comparison irrelevant. (For this exploration, ΔJPF does not even use the optimized state comparison, from section 4.2.4.) As mentioned earlier, JPF is a general-purpose model checker that stores and restores the entire Java states and thus has a high execution and backtracking overhead.

In BOX, ΔExecution sometimes results in a higher code execution time, yet has a smaller overall exploration time. The reason is that ΔExecution achieves significant savings in the state comparison using the optimized algorithm from Section 4.2.4. For example, for `bst` and $N = 11$, ΔExecution increases the execution time from 3.26s to 7.96s. However, it reduces the state comparison time from 57.19s to 20.35s, which more than makes up for the longer execution time. Note that the number of states and state comparisons is the same in both standard execution and ΔExecution, but the optimized state comparison is only possible for ΔExecution. Indeed, it is the execution on ΔStates that enables the simultaneous comparison of a set of states.

Memory

Tables 4.3 and 4.4 also provide a comparison of memory usage. Specifically, the columns labeled "mem. std/delta" show the ratio of peak memory usage for standard execution versus ΔExecution. Our setup uses the Sun's `jstat` [Sun] monitoring tool to record the peak usage of garbage-collected heap in the JVM running an experiment. Although this particular measurement does not include the entire memory used by the JVM process, it does represent the most relevant amount used by a model checker. (The cells marked "-" represent experiments where the running time is so short that `jstat` does not provide accurate memory usage.)

For JPF, standard execution uses more memory than ΔExecution for most experiments. The results show that ΔExecution reduces memory use from 0.46x to 11.50x (with median 1.48x). Note that ΔExecution occasionally uses more memory, for example for `bst`. In BOX, ΔExecution reduces memory from 0.18x to 2.71x (with median 1.18x). Note that the median of memory use in BOX has a lower value indicating that ΔExecution consumes more memory (relative to the

experiment		standard execution time			ΔExecution time			
subject	N	exec.	comp.	backt.	exec.	comp.	backt.	merg.
binheap	7	0.22	0.37	0.12	0.10	0.13	0.00	0.06
	8	4.57	3.74	2.78	1.01	1.43	0.01	0.87
	9	21.46	67.74	15.03	4.70	21.77	0.01	6.27
bst	9	0.23	1.90	0.18	0.47	0.73	0.01	0.29
	10	0.52	10.60	0.91	1.78	3.78	0.01	1.86
	11	3.26	57.19	4.42	7.96	20.35	0.02	21.14
deque	8	0.32	1.50	0.33	0.15	0.39	0.00	0.19
	9	2.30	16.26	3.17	1.36	4.46	0.00	1.57
	10	21.95	214.30	31.48	16.48	59.01	0.01	23.87
fibheap	6	0.06	0.08	0.03	0.05	0.04	0.00	0.03
	7	0.31	0.51	0.29	0.21	0.24	0.00	0.18
	8	4.70	7.94	3.90	2.77	3.76	0.00	3.18
filesystem	3	0.02	0.09	0.01	0.04	0.06	0.00	0.03
	4	0.06	0.99	0.06	0.16	0.38	0.01	0.06
	5	3.30	30.35	1.70	16.74	10.45	0.02	2.59
heaparray	8	0.12	0.88	0.12	0.38	0.35	0.00	0.10
	9	1.17	9.25	1.06	3.73	4.05	0.00	1.03
	10	11.47	98.01	10.46	44.85	46.36	0.01	18.52
queue	6	0.05	0.19	0.10	0.03	0.05	0.00	0.04
	7	0.80	1.85	1.09	0.07	0.45	0.00	0.39
	8	13.71	42.80	21.38	0.94	9.77	0.00	14.57
stack	6	0.04	0.16	0.08	0.02	0.04	0.00	0.03
	7	0.40	1.54	0.94	0.02	0.34	0.00	0.29
	8	7.04	34.77	16.58	0.02	7.54	0.00	10.21
treemap	12	1.44	29.60	1.26	1.55	6.74	0.02	0.66
	13	4.23	94.53	4.05	4.39	22.04	0.02	2.42
	14	13.48	333.97	13.08	13.43	81.55	0.04	9.13
ubstack	8	0.22	1.68	0.24	0.36	0.70	0.00	0.16
	9	2.64	16.96	1.62	3.94	7.96	0.00	1.54
	10	33.77	203.66	16.28	50.82	100.10	0.00	22.12

Table 4.4: Time breakdown for BOX experiments.

standard execution). This is justified by the fact that the ΔExecution implementation in JPF uses native states in some parts. For these parts, memory management is done by the base-level JVM. In constrast, in standard execution, only one JVM – the JPF's JVM – handles the memory management.

Many factors, already mentioned for exploration time, can influence the memory usage, but an important factor seems to be the number of constant ΔObjects in the merged state, i.e., the ΔState. ΔExecution uses these objects to represent values that are the same across all states in a ΔState. To conduct this experiment, we calculate the percentage, across all iterations, of the number of objects on the merged state that are actually constant. For example, if we run an experiment for 2 iterations and find x_1 constants out of y_1 delta objects in the first iteration and

experiment		standard execution			ΔExecution			ratio
subject	N	#states	#exec.	tot.time	#states	#exec.	tot.time	tot. impr.
binheap	28	28	15680	4.33	28	956	4.12	1.05x
	29	29	16820	4.42	29	958	4.16	1.06x
	30	30	18000	4.58	30	1040	4.27	1.07x
bst	20	166064	10168360	549.85	150192	49645	90.86	6.05x
	21	381535	22466178	1,237.36	416946	77951	246.28	5.02x
	22	677848	43605496	2,389.23	626555	83569	380.42	6.28x
fibheap	28	881	182323	18.68	1041	7810	20.40	0.92x
	29	961	184320	19.15	1157	7269	20.35	0.94x
	30	1144	289571	28.68	1354	10981	28.56	1.00x
treemap	20	11879	1492080	195.50	11952	39131	43.28	4.52x
	21	22455	2893212	385.33	20590	48974	65.82	5.85x
	22	38126	4918100	661.17	36550	59693	107.33	6.16x

Table 4.5: Overall time for non-exhaustive exploration.

x_2 out of y_2 in the second, we report $(y_1 + y_2 - x_1 - x_2)/(y_1 + y_2)$ as the percentage of constants. It is important to mention that this measurement of constant is different than the one reported on Figure 4.2. This measure reflects more precisely the use of memory, while the other, appearing in the charts from Figure 4.2, illustrates better the appearance of constants during execution.

We found that there is a relatively strong positive correlation between the percentage of constant ΔObjects and the memory ratio for an experiment. For example, `bst` and $N = 11$ has a poor memory ratio, and the percentage of constant objects in ΔStates is 33%, the lowest of all subjects. For `treemap` and $N = 12$, on the other hand, ΔExecution uses less memory than standard execution, and the percentage of constant objects is 69%. These percentages are computed across the entire exploration: it is the percentage of constant delta objects produced during merging from the total number of delta objects produced.

4.4.3 Non-exhaustive exploration

We next evaluate ΔExecution for a different state-space exploration. While exhaustive exploration is the most commonly used, there are several others such as random [PE05, CS04] or symbolic execution [KPV03, XMSN05, dPX+06]. Recently, Visser et al. [VPP06] have proposed *abstract matching*, a technique for non-exhaustive state-space exploration of data structures. The main idea of abstract matching is to compare states based on their *shape abstraction*: two states that have the same shape are considered equivalent even if they have different values in nodes. For example, all binary search trees of size one are considered equivalent. The exploration is pruned

whenever it reaches a state equivalent to some previously explored state, which means that abstract matching can miss some portions of the state space.

We chose to evaluate ΔExecution for abstract matching because the JPF experiments done by Visser et al. [VPP06] showed that abstract matching achieves better code coverage than five other exploration techniques, including exhaustive exploration, random, and symbolic execution. (The experiments did not consider whether higher code coverage results in finding more bugs.) Our evaluation uses the same four subjects used to evaluate abstract matching in JPF: `binheap`, `bst`, `fibheap`, and `treemap`. We ran each subject for sequence bound up to $N = 30$ (as done in [VPP06]) or until the experiment timed out of 1 hour. We used the same drivers as for exhaustive exploration but randomized the order of non-deterministic choices in `getInt` and used 10 different random seeds; Visser et al. use the same experimental setup to minimize the bias that a fixed order of method/value choices could have when combined with abstract matching.

Table 4.5 shows the results for abstract matching with and without ΔExecution. ΔExecution significantly reduces the overall exploration time for two subjects (`bst` and `treemap`) and slightly reduces or increases the time for the other two subjects (`binheap` and `fibheap`). ΔExecution provides a smaller speedup for the bounds explored for abstract matching (Figure 4.5) than for the bounds explored for exhaustive exploration (figures 4.1 and 4.2). This can be attributed to the reduced number of states and executions in abstract matching compared to exhaustive exploration. For example, for `bintree`, abstract matching for $N = 20$ explores fewer states and executions (166,064 and 10,168,360, respectively) than exhaustive exploration for $N = 11$ (915,641 and 20,144,102). In addition, there is less similarity across states and executions in abstract matching than in exhaustive exploration. Indeed, abstract matching selects the states such that they differ in shape. (The peculiarity of `binheap` is that it has only one possible shape for any given size.)

Note that abstract matching can explore a different number of states and executions with and without ΔExecution. The reason is that standard execution and ΔExecution explore the states in a different order: while standard execution explores each state index in order, ΔExecution explores at once various subsets of state indexes based on the splits during the execution. Thus, these executions can encounter in different order states that have the same shape, and only the first encountered of those states gets explored. The randomization of non-deterministic method/value

choices, which is necessary for abstract matching, also minimizes the effect that different orders could introduce for ΔExecution and standard execution. As Figure 4.5 shows, ΔExecution can explore more states (for example for `bst` and $N = 21$) or fewer states (for example for `bst` and $N = 20$) than standard execution, but ΔExecution speeds up exploration whenever the shapes have similarities.

4.4.4 AODV case study

We also evaluated ΔExecution on a larger application, namely the implementation of the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [PR99] in the J-Sim network simulator [J-S]. This application was previously used to evaluate a J-Sim model checker [SVMH05] and Mixed execution as discussed in Chapter 3.

AODV is a routing protocol for ad-hoc wireless networks. Each of the nodes in the network contains a routing table that describes where a message should be delivered next, depending on the target. The safety property we check in this study expresses that all routes from a source to a destination should be free of cycles, i.e., not have the same node appear more than once in the route [SVMH05].

The implementation of AODV, including the J-Sim library classes that it depends on, consists of 43 classes with over 3500 non-blank, non-comment lines of code. We instrumented this code using the Eclipse plug-in that automates instrumentation for ΔExecution on JPF. The resulting instrumented code consisted of 143 classes with over 9500 lines of code. We did not try this case study in BOX since it currently requires much more manual work for instrumentation.

We used for this case study the driver previously developed for AODV [SVMH05]. Like the `bst` driver shown in Figure 4.1, the AODV driver invokes various methods that simulate protocol actions (sending messages, receiving messages, dropping messages etc.). Unlike the `bst` driver, the AODV driver (i) includes guards that ensure that an action is taken only if its preconditions are satisfied and (ii) includes a procedure that checks whether the resulting protocol state satisfies the safety property described above. Section 2.7 discusses the AODV protocol in more detail and Section 3.4.3 describes the events and their guards used in the AODV driver. In this experiment, when a violation is encountered, the driver prunes that state/path but continues the exploration.

| experiment | | JPF time | | | JPF mem. | # states |
subject	N	std	delta	std/delta	std/delta	
	6	6.87	7.81	0.88x	0.53	1061
	7	21.44	16.97	1.26x	0.56	3796
aodv	8	74.31	43.10	1.72x	0.52	13195
	9	262.20	128.60	2.04x	0.58	44735
	10	926.60	485.14	1.91x	0.51	147805

Table 4.6: Exploration of AODV in JPF.

| experiment | | standard execution time | | | ΔExecution time | | | |
subject	N	exec.	comp.	backt.	exec.	comp.	backt.	merg.
	6	3.21	0.20	3.46	4.82	0.54	1.93	0.53
	7	11.48	0.64	9.32	11.79	1.96	2.28	0.94
aodv	8	41.72	2.47	30.11	29.57	7.76	3.39	2.38
	9	148.06	9.51	104.63	85.88	29.68	6.00	7.04
	10	522.49	36.18	367.92	337.67	110.65	14.46	22.36

Table 4.7: Breakdown of the exploration time for AODV in JPF.

We ran experiments on three variations of the AODV implementation, each containing an error that leads to a violation of the safety property [SVMH05]. Table 4.6 shows the results of experiments on one variation. Since the property was first violated in the ninth iteration for all three variations, the results for the other two variations were similar, and we do not present them here. It is important to point out that we continue the exploration after encountering a bad state.

Table 4.7 shows the breakdown of time for the AODV experiments that appears on Table 4.6. Note that the bulk of the time in ΔExecution goes to the execution operation. The next section elaborates on how the technique can improve the overall time ratio of the AODV experiment.

Optimizations

We implemented two optimizations in the evaluation of AODV. The first takes advantage of pre and post-conditions of methods that the driver for AODV uses to explore the AODV state. The second takes advantage of domain-specific knowledge about AODV: some data-structures on the AODV state are semantically a set. For example, it does not matter in which order a routing tables for an AODV node stores its entries. Below, we provide more detail about these optimizations.

More efficient exploration. We realized that the model checker exercised an AODV transition (say, a message loss) more than once in a given iteration of the Δ-driver. That happens because the evaluation of the pre-condition splits execution. As result, the model checker executes

the same transition separately with different state masks; reducing the potential of ΔExecution to take advantage of the similarity across states and paths. A similar scenario affects the evaluation of the post-condition: the execution of transition splits the state (masks). To improve the exploration we changed the driver to evaluate the pre-condition (respectively, post-condition) in breadth-first order and to merge the state masks at the end of the evaluation. This way the model checker executes a transition only once (in a given iteration) against all states that evaluate the pre-condition to true.

Domain-specific knowledge. We realized that some of the data-structures that the AODV implementation uses are conceptually sets implemented with lists. As result, the model checker can explore more states than it needs. For instance, the model checker will render two states different even if they only differ in the order of the elements in these lists. We realized that the routing table is one of such cases. We changed the implementation to keep the routing tables sorted. This resulted in less states explored in both standard and ΔExecution.

4.5   Limitations and Future Work

We now list the limitations of ΔExecution (and its current implementation) as well as plans for future work.

Supporting merging in different program points. ΔExecution currently supports only merge of state at the boundaries of method calls. The model checker uses a test driver to perform bounded-exhaustive exploration and merge all new states that execution creates during one iteration. This scenario enables our implementation to merge *only* the state of the subject under test. We did not evaluate how effective merging in different program points would be to increase sharing and improve ΔExecution. For example, one could decide to merge states after executing two different conditional branches in separate. (This is similar to the application of a join in abstract interpretation [CC77].) In principle, it is possible to merge state at any program point but that requires creating more deltas of state. For example, if ΔExecution merges state when the program stack is not empty, it needs to create delta state for the program stack.

ΔExecution with different search strategies. Currently, ΔExecution works with a breadth-first search strategy. In principle, the model checker can apply ΔExecution with other search

strategies. Like the application for the verification of concurrent programs, this extension also requires merging state at arbitrary points. For example, the model checker can merge state in a depth-first search after a fixed number of non-deterministic choices. It remains to investigate how ΔExecution can operate with different search strategies. In particular, how these changes could impact usage of constants and ultimately the benefit of ΔExecution.

DeltaExecution with non-primitive inputs and with sets of inputs. Currently, the methods the bounded-exhaustive exploration exercises in our experiments either do not take non-primitive arguments or take non-primitive values that requires only a primitive for their construction (e.g., `java.lang.Integer` object). We consider the construction of non-primitive input values as an orthogonal problem.

Our scenario of exploration currently represents the state of the receiver with a ΔObject (i.e., a set of heaps), but it passes *single values* as arguments to the method call. An alternative to this is to pass sets of values as argument. This would potentially increase the overlapping in ΔExecution: it could be possible that several (individual) executions take the same path for different input arguments. Note, however, that this would require ΔExecution to support *cartesian products*. The state indices of the ΔState of the receiver argument does *not* relate to the state indices of the input arguments. This alternative shows a tradeoff between space and time efficiency in program model checking. More specifically, a tradeoff between memory and time efficiency in ΔExecution.

ΔExecution for concurrent programs. We want to investigate the application of ΔExecution for model checking concurrent programs. This application will require to represent more parts of the state as deltas. In particular, it would require to create deltas for the program stack and the synchronization monitors. In this application, there is no restriction on where the scheduler will preempt execution of one thread. Therefore, it needs to represent the differences on the state of the program stack. In addition, the different interleaving of the program can result in different states of the synchronization monitors.

ΔExecution for path validation and regression testing. The key idea of ΔExecution is to share the commonalities between different program executions. We use the technique to perform bounded-exhaustive exploration with the goal of optimizing time of explicit-state model checkers. We have been investigating the use of ΔExecution in a different scenario, a scenario with fewer paths

and states. In this scenario, we run different versions of a program with the same initial state. The goal is to speed up these executions. Execution creates deltas of states from the differences *in the program* (or the subsequent deltas of state). We consider two applications that follow this scenario: patch validation [ZMS$^+$07] and regression testing [ECDD06]. In patch validation, ΔExecution applies to one pair of program versions whose executions are long-running. In regression testing, ΔExecution applies to a pair of program versions whose executions are short-running. In either case, the motivation is to "validate" the execution of the new version by comparing the state it creates with the state the original version creates. The goal is to speed up the validation process.

ΔExecution using other representation of (set of) state(s). We discussed in the introduction the relationship between symbolic model checking [CGP99, JEK$^+$90] and ΔExecution. Symbolic model checking inspired ΔExecution. Conceptually, ΔExecution performs the same exploration. In contrast to traditional implementations of symbolic model checking, our implementation of ΔExecution handles states that involve heaps.

Symbolic model checking use BDDs as a data structure to implement boolean functions. BDDs have been also used in program analysis [WL04, LH04] to represent analysis information as sets and relations. In particular, to represent approximations of set of heaps. These techniques employ either data [LH04] or control abstractions [WL04] to produce over-approximations of the set of reachable program heaps. Such abstraction makes the analysis of programs with unbounded number of heap states tractable. In contrast to these techniques, ΔExecution does not perform any form of approximation in the state set. It runs the program and accumulates the actually reachable heaps.

We want to investigate other symbolic representations, such as BDDs, to represent sets of concrete heaps in ΔExecution. We discuss next one possible approach. This approach partitions a ΔState in two parts: the concrete part and the symbolic part. The concrete part includes only the regular objects of the state. The symbolic part includes only the ΔObjects. For example, we partition the ΔState appearing in Figure 4.3 in two. The fields of regular objects store symbolic names to express that it can in fact refer to multiple values. BDDs, or other symbolic representation, represent the state of ΔObjects. Note that we do not use indices to enumerate state.

Figure 4.11 shows an illustrative example of such encoding. It highlights some operations that arise during ΔExecution. It shows first the merging of two trees. Then, it shows a split on a

predicate that checks whether the left field of the tree is `null`. Following the split, the figure then shows an assignment of `null` to the left field on one path and an assignment of the right field to the left field on the second path.

In each step, the figure shows, for the sake of illustration, three distinct representations of state. The first is the representation of an object graph similar to the one we currently use in ΔExecution. The second represents state with a boolean formula. We use the literals $NULL$, a, b, and c to denote memory locations. The third representation uses BDDs. Note that we require two booleans to encode each of the heap variables $x0$, $x1$, $x2$, $x3$; we use $z0$ and $z1$ to encode $x0$, $z2$ and $z3$ to encode $x1$, $z4$ and $z5$ to encode $x2$, and $z6$ and $z7$ to encode $x3$.

We use disjunction of boolean formulas to implement merging. BDDs [CGP99] can efficiently implement disjunction, as well as any of the standard boolean operations. We use conjunction of boolean formulas to implement splitting, i.e., we add a new formula to constrain the set of states it represents. We model assignment also with conjunction but that requires introducing a new variable. For example, we introduce a new variable $x3$ to hold the value of the assignment.

Note that this approach requires a large number of symbolic variables that execution can use. Note also that execution can garbage collect parts of the state. For example, the post-state of the execution that assigns `null` to the left field from root cannot reach the state referred to the symbolic name $x1$. (Note that the boolean variables that encode $x1$ appear on the BDD.)

Whaley and Lam [WL04] show that it is possible to describe a variety of pointer analysis algorithms using a declarative, relational language. They describe a system that can translate programs in such language into efficient analyzers based on BDDs. In one of their examples, they model the effect of variable assignment on the results of the analysis with a construction that involves a combination of standard relational operations: projection, renaming, natural join, and union. In contrast to our application, heap updates are not destructive: the analysis approximates the solution iteratively; it typically does not remove tuples on the sets. The destructive update was the reason for creating different names in the approach above.

We want to study other encodings of state with heaps and evaluate their use in ΔExecution. We also want to investigate extensions of BDDs that support arrays natively or enables the efficient encoding of arrays; the model checker can use arrays to represent execution state.
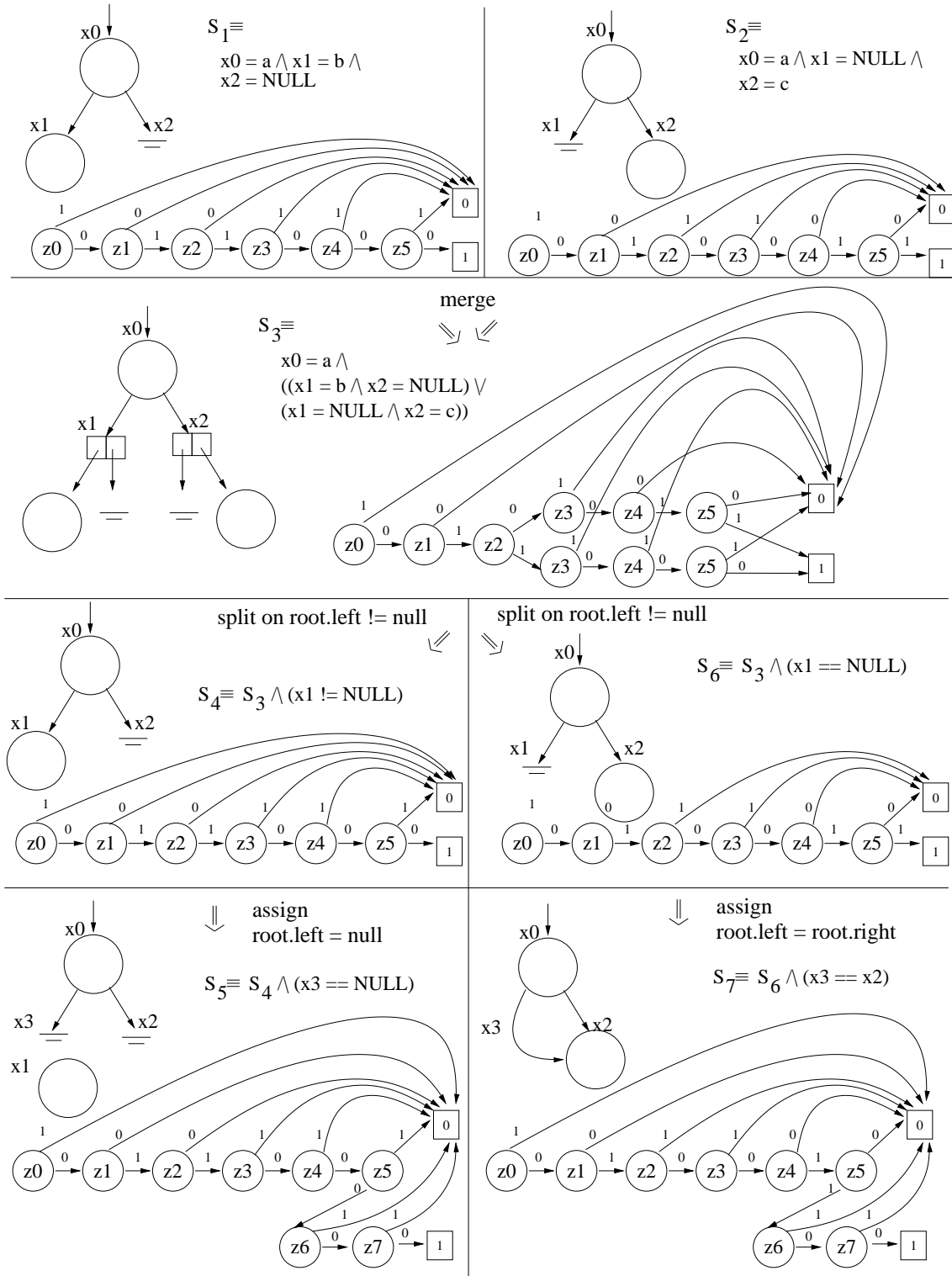
Figure 4.11: Use of boolean functions to encode set of heaps. Each object graph appears with a boolean formula and a BDD that encodes it. The top part of the figure shows two states of a tree (with their different representations). The second part shows the merging of these two states. The last two parts show a split followed by a field assignment.

Chapter 5

Related Work

Traditional model checkers such as Murphi [DDHY92], SMV [McM93], and SPIN [Hol97] have been extensively used in formal reasoning of both hardware and software systems. These tools analyze models written in special modeling languages. To analyze a system, the user needs either to manually write a model of the system in a language understood by the tools [BOG02] or to automatically translate an implementation of the system from a programming language (e.g., Java) into the modeling language of the tools [PSSD00, Hav99, CDH$^+$00]. Our work considers model checkers that directly analyze systems written in a programming language.

Software and hardware often produce models with infinite or finite but extremely large number of states. The problem of models having large number of states is known as the *state space explosion problem*. While state space explosion is a key issue in model checking, *time efficiency* becomes more relevant with the increasing appearance of model checkers that operate on actual code. State comparison and backtracking, which are two of the major operations in explicit-state model checking of programs, become potentially slower: the state of an implementation model is larger as it includes details not present in a design model (of the same system). In addition, a deterministic step of the exploration is also potentially more expensive: a deterministic step of an implementation model typically contains more state transitions than one in a design model (of the same system).

We next discuss related work to this dissertation. We separate the body of related work in two groups. The first group includes related work with similar goal – speed-up the state space exploration of programs – and the second group includes related work that employs similar technique to the ones discussed in this dissertation.

5.1 Work with similar goal

This section discusses related work that has similar goal with our techniques. We divide this section in two groups. The first group includes techniques that target the *reduction of space*. The second group includes techniques that target *reduction of time*.

5.1.1 Space-reduction techniques

A variety of techniques have been developed with focus on reduction of space, i.e., the reduction in the size of the states and/or the number of transitions in the input model. They have been successful in reducing the impact of the state space explosion problem and ultimately making model checking more practical. Important to note is that such techniques can help to reduce the exploration time: the model checker have less states to explore; it takes less time to execute transitions, backtrack, and compare states. These techniques are orthogonal to the ones we described in this dissertation. As such, it is possible to apply space-reduction techniques *in conjunction* with ours. In fact, the evaluation of our techniques used some of these techniques: heap symmetry (in mixed and ΔExecution) and abstract matching (in ΔExecution).

Space-reduction techniques typically fall into two categories: those that apply some form of abstraction in the input program prior to model checking and those that prune paths from the state space.

Abstraction

This section discusses techniques that reduce the size of the model by performing some form of abstraction. We call *abstraction* a transformation on the input model that produces another model that is simpler to verify: the model checker may contain less information in states or less state transitions.

Predicate abstraction [VPP00,BR00,BMMR01] is a technique that transforms general programs into boolean programs which are amenable to efficient verification with BDDs. While predicate abstraction has shown great results in many applications, it does not handle well complex data structures and heaps. (See sections 2.3.1 and 4.5.) ΔExecution handles heaps efficiently. However, ΔExecution does not perform any abstraction on the input program that could potentially reduce

space and time.

Program slicing [Wei81, HCD$^+$99] is a technique for simplifying a program with respect to some property of interest. It takes as input a program and a description of a property and produces as output another program that must only include the dynamic behaviors of the original program that affect the property of interest, i.e., it can remove the parts that cannot affect the property. Applications of program slicing include software testing and debugging, e.g., a debugger can apply program slicing to help the programmer focus her attention to only the parts of the program that affect the error that she is trying to locate [ADS93]. Slicing can reduce the size of the model as it can remove data from the state and state transitions not relevant for the checking [HCD$^+$99, HDZ00]. We want to investigate the impact of mixed execution and ΔExecution on model checking of sliced programs.

Path pruning (state comparison)

This section discusses techniques for path pruning, i.e., techniques that identify and eliminate unnecessary program paths from the state space exploration.

Symmetry reduction is a space-reduction technique that exploits the fact that software components often play similar roles in a system. The model checker can prune a path when it leads to a state that is *symmetric* to one already visited. Thread symmetry [HJJJ85, CEFJ96, ES96, ID96] is a special kind of symmetry that often occur in concurrent systems. For instance, consider the usual dining philosophers problem [Dij71] with a fixed number of philosophers. The state in which one philosopher, say i, eats and all other philosophers think is equivalent to the state in which another philosopher $j \neq i$ eats and all others, including philosopher i, think. Such two states are symmetric since there is only one role for a philosopher in the algorithm. We say that the states are equivalent under permutation of the process/philosopher ids. We want to apply symmetry reduction for ΔExecution of concurrent programs.

Heap symmetry [MD05, LV01, CGP99, Ios01] is an important technique that model checkers use to alleviate the state space explosion problem. A model checker uses heap symmetry to detect equivalent states during the state space exploration: the model checker can prune a path when it leads to a state equivalent to some already visited. In object-oriented programs, two heaps

are equivalent if they are *isomorphic* (i.e., have the same structure and primitive values, while their object identities can vary) [Ios01, MD05, BKM02]. An efficient way to compare states for isomorphism is to use *linearization* (also known as serialization or marshalling) that translates a heap into a sequence of integers such that two heaps are isomorphic if and only if their linearizations are equal. It is important to note that the current implementation of ΔExecution applies heap symmetry only to heaps with a single root.

Partial-order reduction [KP89, Val91, GS92, God96] is a space-reduction technique that model checkers often apply to optimize verification of (concurrent) programs. The technique exploits the commutativity of transitions in the model; some sequence of transitions can lead to the same state when executed in different orders. For instance, assume transitions a and b commute (e.g., transitions a and b mutate different fragments of data in the program and there is no flow of data between these fragments), and whenever the model checker explores a, it can explore b from the post-state and vice versa. Then, it is possible to prune the paths from a followed by b without loss of generality as the model checker explores the paths from b followed by a, which lead to the same state. We want to apply partial-order reduction to improve state space exploration of programs.

Visser et al. [VPP06] recently proposed *abstract matching*, a non-exhaustive technique (see Section 4.4.3) for state space exploration of dynamically-allocated data structures. The main idea of abstract matching is to compare states based on their *shape abstraction*: the model checker considers two states equivalent if they have the same shape; the model checker does *not* take into account the values these states associate to primitive fields. For example, the model checker with abstract matching considers as equivalent all binary search trees of size one. It prunes the exploration path whenever it reaches a state equivalent to some previously explored state. Note that the abstract matching can result in a more efficient exploration as the model checker misses portions of the state space. We evaluated ΔExecution with abstract matching and realized a decrease in the speedup when compared to ΔExecution without abstract matching. The relative decrease of improvement in ΔExecution under this scenario is due to the fact that abstract matching reduces the total number of states the model checker explores. (See details in Section 4.4.3.) We did not yet apply abstract matching for subjects that contain bugs. It remains to evaluate how often abstract matching miss paths that lead to a bug.

Darga and Boyapati proposed glass-box model checking [DB06] for pruning search. They proposed a static analysis that can reduce state space without sacrificing coverage. Glass-box exploration represents the search space as a BDD and identifies, without executing code, parts of the state space that would not lead to more coverage. However, glass-box exploration requires the definition of executable invariants in order to guarantee soundness. In contrast, ΔExecution does not require any additional annotation on the code.

5.1.2 Time-reduction techniques

We next discuss work related to the reduction of time in the major model checker operations: execution, path exploration, and path pruning.

The application of space-reduction techniques provides a way to reduce overall exploration time as the model checker explores less states. It is important to note, however, that model checkers can reduce exploration time without reducing the size of the state space. Improving *time efficiency* becomes more relevant with the increasing importance model checkers for programs and also with the advances in space-reduction techniques. This section discusses related work targeting the reduction of time itself, not space.

Recall from Section 2.5 that handling state is central to explicit-state model checkers [Hol97, LV01, Ios02, MD05]; state representation is a concern that crosscutts different parts of a model checker. Designers of software model checker need to decide (i) whether the model checker uses native states or not for the execution of the subject program; (ii) whether it applies re-execution, copying of state, or restoration of state to perform path exploration; and (iii) whether or not it uses state comparison in order to prune paths. For example, Verisoft [God97] uses native states, re-executes code, and does not compare state (it only applies partial-order reduction to prune paths that lead to same states); CMC [MPC+02] uses native state, copies the state (it uses operating system support to fork execution; it uses "copy on write" to only copy the parts of the state prior to a write during the execution of a deterministic step), and compares states; JPF uses a special representation of state, restores the parts of the state that a deterministic step mutates, and compares states (it uses both partial-order reduction and heap symmetry).

Execution

We use the term execution to refer to the operation that carries out a deterministic step. Some software model checkers use native representation of state in order to speed up execution. With native states, the model checker executes code using the same environment as a regular execution. In the case of C, the model checker runs the compiled code for the program on top of the operating system. In the case of Java, the model checker runs the code on top of a regular JVM. Examples of model checkers that follow this design include Verisoft [God97], CMC [MPC$^+$02, ME04, YTEM04], and BOX (see Section 4.3.2). Verisoft was the first model checker to directly analyze the implementation code, specifically code written in the C language; CMC has been used to model check Linux implementations of networking code (e.g., AODV and TCP) and file systems; and BOX is a model checker for sequential Java code used to evaluate ΔExecution.

It is important to note that this approach does not offer to the model checker a unified view of the memory. For example, CMC does not control the locations of the heap objects that a program execution allocates. It reads state by following roots of heap objects which are available to a test driver. A unified view of memory can be important to the efficient implementation of some memory-intensive model checking operations, such as state comparison or backtracking. JPF follow this approach. It represents the heap as a single array of integer arrays, which we call the *special representation of state*. It can compare two instances of such arrays (or fragments of the arrays), and efficiently index into this array. It is important to note that this special representation can sacrifice execution as the model checker needs to keep track of all field writes so to be able to recover state on backtracking. Mixed execution circumvents this problem for fragments of code that are deterministic.

In summary, native state representation can be helpful to execute transitions, special state representation can be helpful for defining memory intensive operations such as state comparison and backtracking. Mixed execution explores one way for the model checker to combine different state representation without a major sacrifice in performance of key model checking operations (See Section 3). We remain to investigate other alternatives. For example, it remains to evaluate how native execution and special representation of state can work together to speed up both execution and the overall state space exploration. (See more details in Section 3.5.)

102

Path exploration (backtracking)

Model checkers perform backtracking in many different ways. Verisoft [God97], for example, re-executes code to explore a different exploration path. Other model checkers can restore state on backtracking. For example, BOX creates "undo" objects. Each "undo" object stores the value of a field prior to its mutation along the execution. This allows BOX to restore the entire state by applying sequentially field modifications, i.e., "undoing" the changes. Yet another option is to copy part of the state when performing a write in the state. For example, CMC [MPC+02] uses the support of the operating system to perform copy-on-write. In this approach, state restoration is not necessary because the model checker (in the case of CMC, using the operating system support) copies the data when it identifies that execution is about to mutate state that is visible to a different execution. (See more details in Section 2.5.)

ΔExecution reduces, in both JPF and BOX implementations, the total time that backtracking takes in the state exploration. The reason is that ΔExecution incurs in a smaller number of backtrackings. For the JPF implementation, backtracking took a significant time of the exploration (see Section 4.3) and the benefit of this reduction was more noticeable. In contrast, backtracking in BOX did not consume a significant fraction of time (see Section 4.4) from the overall exploration; the reduction of backtracking time for ΔExecution was thus smaller.

It is important to mention that mixed execution does not impact backtracking, and the impact of ΔExecution on backtracking is *not* related to an improved backtracking mechanism. The improvement is due to the reduced number of backtrackings. Path exploration is related to our goal of improving overall state space exploration but is orthogonal to mixed execution and ΔExecution. The techniques we described can work with different backtracking mechanisms.

Path pruning (state comparison)

One particular way to prune paths from the exploration is to compare states that arise during the search: the model checker can prune a path that leads to a state that was already visited. (See Section 5.1.1.) The state comparison operation is typically responsible for a significant fraction of the total exploration time.

JPF and BOX compare states using Iosif's depth-first heap linearization algorithm [Ios01]. The

implementations of ΔExecution in these model checkers compare states using the same algorithm. ΔExecution leverages the fact that set of states can be explored simultaneously to produce a set of linearizations. We show in sections 4.3 and 4.4 that the algorithm that operates on sets performs much more efficiently than the one that performs the linearization on one-by-one state from the set. The main reason is that the optimized algorithm shares the prefixes of linearization that several states have in common. It is important to note that we did not improve Iosif's algorithm.

Musuvathi and Dill proposed an algorithm for incremental state hashing based on a breadth-first heap canonicalization [MD05]. Their algorithm strives to preserve object identities when execution mutates the object graph. (When using the depth-first algorithm, a small change in the graph can result in a severe change in object identities.) In addition, the computation of the hash is incremental: a model checker can compute the hash of the entire graph by composing the hashes of individual objects. Such algorithm allows to reuse hashes for the parts of the trees that did not change (as it tries to preserve object identities) and recalculate efficiently new hashes for the parts that did change (as it is compositional). We plan to implement this algorithm in JPF and to use ΔExecution to optimize state comparison even further. Again, note that ΔExecution does not optimize the algorithm itself, but its performance when operating with sets of states.

5.2   Work with similar technique

This section discusses work that uses a technique that is similar to the ones discussed in this dissertation.

5.2.1   Abstract Interpretation

Abstract interpretation [CC77] is a theory of approximations of the program semantics based on ordered sets; more precisely, on lattices. Two major applications of abstract interpretation are program optimization and debugging. The idea of an abstract interpretation is to execute the program over an abstract domain. Such kind of execution introduces imprecision but can often prove (simple) properties of a program.

Consider, for example, that the user is interested on properties about the sign of values of program variables. To that end, the user defines the abstract domain of sign $\mathcal{P}\{-, 0, +\}$ that the

program execution will use. Now, consider the abstract interpretation of the assignment `c = a * b` where the variables `a` and `b` store, respectively, the abstract values $\{-, +\}$ and $\{-\}$. The post-state then associates $\{-, +\}$ to variable `c`; this abstract value is the most precise value of `c` that results from the multiplication. (Note that the interpretation manipulates sets and that it results in less computation; the analyzer needs only to compute the resulting sign of the arithmetic operation.) The abstract interpretation of a program sometimes requires to accumulate information that flows from different program paths. For example, consider the program `if(...){c=...}else{c=...}` •. The abstract value of the variable `c` at program point • is the "join" of the values that flow from each branch. A join of two elements (in a lattice) is the least upper bound (i.e., the minimum abstract value that is superior to both elements) of these elements.

ΔExecution also merges sets of states. In this sense, merging is similar to the use of join in an abstract interpretation that uses a partial-order whose elements are sets, and the order relation is set inclusion. However, it is important to highlight that ΔExecution does not merge (/join) states in the same way that abstract interpretation. For example, ΔExecution does not merge states after the execution of conditional branches but only at method boundaries in a bounded-exhaustive exploration of the test subject. Merging state in other program points potentially requires to create more deltas of state. (See details in Section 4.5.)

5.2.2 Symbolic Model Checking

In contrast to explicit-state model checking, symbolic model checking represents states implicitly. It represents state and the transition relation as boolean functions. Symbolic model checking performs by finding fixpoints of these functions. (See Section 2.3.)

As previously discussed in section 2.3.2 and 4.5, the implicit encoding of states with heaps can be problematic for the *execution* of program transitions, which is an important operation in program model checking (see Section 2.5). ΔExecution provides a way to dealing with sets of concrete heaps. It proposes a data-structure that encodes a *set of concrete heaps* and enables *fast execution* of transitions. In addition, this data-structure provides operations on sets namely, union (merge) and disjoint partition (split), which enables efficient overall exploration. For that reason, we consider ΔExecution as a technique that performs symbolic model checking on heaps.

5.2.3 Symbolic execution

Symbolic execution [KPV03, VPK04, XMSN05] is a special kind of execution that operates on symbolic values. The state in a symbolic execution includes symbolic variables (that can represent a set of concrete values) and a path-condition that encodes constraints on the symbolic variables. Symbolic execution has recently gained popularity with the availability of fast constraint solvers and has been applied to test-input generation for object-oriented programs [KPV03, VPK04, XMSN05]. In the general case, symbolic execution generates constraints that are undecidable. The recent techniques combining symbolic execution and random execution show good promise in handling some of common problems in symbolic execution such as undecidability of constraints, and unavailability of libraries [GKS05, SMA05, CGP$^+$06]. Conceptually, both symbolic execution and ΔExecution operate on a set of states. While symbolic execution can represent an unbounded number of states, ΔExecution uses an efficient representation for a bounded set of concrete states. The exploration with sets of concrete states avoids generating potentially undecidable constraints.

Symbolic execution deals well with primitive value but can be problematic to encode states with heaps. One way to improve both ΔExecution and symbolic execution is to use a concrete representation of set of heap as in ΔExecution, i.e. to explicitly enumerate all heaps of interest, but to use a symbolic representation for primitives.

Offutt et al. [OJP99] proposed DDR, a technique for test-input generation where the values of variables are ranges of concrete values. DDR uses symbolic execution (on ranges) to generate inputs. DDR effectively splits the domains, using a technique called *domain splitting*, when new constraints are added to the system. DDR requires inputs to be given as ranges, implements a lossy abstraction (to reduce the size of the state space in favor of more efficient decision procedures), and does not support object graphs.

ΔExecution focuses on object graphs and does not require inputs to be ranges, but the use of ranges as a special representation in ΔStates could likely improve ΔExecution even more, and we plan to investigate this in the future.

5.2.4 Shape Analysis

Shape analysis [Rug04, YRS04, KLR02] is a static program analysis that verifies programs that manipulate dynamically allocated data structures. Shape analysis uses abstraction to represent infinite sets of concrete heaps and performs operations on these sets, including operations similar to splitting and merging in ΔExecution. Shape analysis computes over-approximations of the reachable sets of states and loses precision to obtain tractability.

Similarly to shape analysis, ΔExecution uses a representation of state that encodes a set of heap objects. However, shape analysis focuses on static verification of programs; not testing. Its goal is to prove heap properties for programs that manipulate dynamic data. Shape analysis derives verification conditions from the program and attempts to prove the heap property from these conditions. ΔExecution, in contrast, uses set of states for the actual execution of a program. It encodes a bounded number of states in a set and can therefore decide any kind of heap property. It is important to mention that we did not write oracles that check for heap properties such as lack of cycles and lack of aliasing. We used sets solely to speed up the state space exploration.

Chapter 6

Conclusions

Our goal is to advance the state of the art in model checking for the purpose of testing. More specifically, our goal is to *improve the efficiency of explicit-state model checking for programs with dynamically allocated data*. To that end, this dissertation presents two techniques. We call these techniques mixed execution and delta execution. Our thesis is that:

> *Mixed Execution and Delta Execution can be effective for reducing overall exploration time in the explicit-state space exploration of programs that manipulate dynamically allocated data.*

Mixed execution is a technique that speeds up model checkers that use special state representation. Such representations of state can be convenient to perform operations that are intensive on the use of memory, such as backtracking and state comparison, but can sacrifice the execution of transitions. Mixed execution enables model checkers that use special state representation to execute code natively, i.e., using native states. This results in faster execution of transitions and therefore faster overall exploration.

ΔExecution is a technique that speeds up explicit-state model checking of programs by applying symbolic model checking on heaps. The technique performs state-space exploration using sets. The use of sets enables the model checker to take advantage of the overlapping that exist across different executions in a systematic state-space exploration. This results in faster execution of transitions and can also reduce the overall time to perform backtracking and state comparison.

We present next a summary of experimental results, a list of the contributions, and the final remarks for this dissertation.

6.1   Summary of experimental results

We implemented mixed execution in JPF. We evaluate mixed execution on seven subject programs and one large case study. The experimental results on the seven smaller programs show that mixed execution can improve the overall time for state exploration in JPF from 1.01x to 1.73x (with median 1.13x), while improving the time for execution of deterministic blocks from 0.91x to 3.05x (with median 1.64x). We also evaluate mixed execution on a larger case study containing bugs. The exploration time on this experiment reduces from 1.14x to 1.41x (with median 1.23x). It is important to point out that mixed execution reduces the overall exploration time by reducing the execution time for deterministic blocks. Mixed execution does not affect the order of exploration, the number of explored states, or any other aspect of the state exploration.

We implemented Delta Execution in two model checkers, JPF (Java PathFinder) and BOX (*Bounded Object eXploration*) to evaluate the effectiveness of this technique in model checkers with different designs. JPF is a popular general-purpose model checker for Java programs, while BOX is a specialized model checker that we developed for efficient exploration of sequential Java programs. The results show that ΔExecution improves the overall exploration in both tools, but the improvements are due to different factors. The improvement in JPF is mainly due to execution and backtracking, while in BOX it is mainly due to the optimized comparison of state as described in Section 4.2.4.

We evaluate ΔExecution on ten simple subject programs and a larger case study with errors. The results show that ΔExecution improves the exploration time for the smaller programs from 0.88x to 126.80x (with median 5.60x) in JPF and from 0.58x to 4.16x (with median 2.23x) in BOX, while taking from 0.46x to 11.50x (with median 1.48x) less memory in JPF and from 0.18x to 2.71x (with median 1.18x) memory in JPF. We also evaluate ΔExecution, using our JPF implementation, on one larger case study with errors, the AODV routing protocol, where ΔExecution improves the exploration time from 0.88x to 2.04x (with median 1.72x).

6.2   Contributions

We show next a list of the contributions of this dissertation.

109

- Ideas of mixed execution and ΔExecution. We proposed two techniques to improve time efficiency of the key operations of explicit-state model checking: execution, path exploration, and path pruning. (See Section 2.5.) Mixed and ΔExecution can improve execution time, while ΔExecution can, in addition, can improve overall time to perform path exploration and path pruning.

- Proposal of a data structure for implementing sets of heaps with single-roots. The proposed data structure is equipped with operations that allow efficient merging and splitting of sets of concrete heaps (see Section 1.4).

- Implementation of mixed execution in JPF, and implementation of ΔExecution in both JPF and BOX. We modified the JPF model checker to implement the two techniques and implemented BOX, a custom built model checker specialized in sequential code.

- Evaluation on several subjects and a larger case study. We evaluated the techniques on data structures used to evaluate other testing techniques [SM03, CS04, Qad, PE05, XMSN05, DB06, VPP06] and on one larger case study, the AODV routing protocol [PR99].

6.3   Final remarks

We validate our thesis with experiments on several data structures and on a larger case study. Mixed execution and ΔExecution lead to a faster state space exploration for most experiments. The reduction in overall time increases with the increase in the bound the model checker uses to perform bounded-exhaustive exploration. (The bound corresponds to the number of method invocations the test driver generates from the initial state. See Section 1.5.1.) For small enough bounds, the application of the techniques did not result in time reduction. But it is typical to use larger bounds during the state space exploration. For example, in the AODV case study we used to evaluate both techniques, the model checker only finds a property violation with a bound greater than 8.

Mixed execution points out the importance of studying the trade-offs used in state-space exploration for model checking and testing. ΔExecution points out that the use of sets can be effective to reduce time of key operations in explicit-state model checking.

Although the techniques we presented show promising results, more work can be done to improve state-space exploration of programs that allocate dynamic data. Sections 3.5 and 4.5 show directions for improving the techniques. For instance, we plan to investigate how to combine special state representation with native execution. (Note that mixed execution employs native execution only on deterministic blocks.) We also plan to evaluate ΔExecution with other symbolic representations and use it in different contexts. For instance, regression testing, patch validation, and in the model checking of concurrent programs.

References

[ADS93]   Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[AG06]   Cyrille Artho and Pierre-Loic Garoche. Accurate centralization for applying model checking on networked applications. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pages 177–188, Washington, DC, USA, 2006. IEEE Computer Society.

[Ake78]   Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[AQR+04]   Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 484–487, 2004.

[ASB+04]   Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. Jnuke: Efficient dynamic analysis for java. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 462–465, 2004.

[Bei90]   Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[BKM02]   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, 2002.

[BLR02]   Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Sonference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[BMMR01]   Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[Boe91]   Barry W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, 1991.

[BOG02]   K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, July 2002.

[BR00]   Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN)*, pages 113–130, 2000.

[Bry92]   Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BW96]   Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[CC77]   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 SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CCGR99]   Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 495–499, London, UK, 1999. Springer-Verlag.

[CDH+00]   James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.

[CEFJ96]   E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.

[CGH+95]   Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[CGP99]   Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[CGP+06]   Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[CKL04]   Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176, 2004.

[CS04]   Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software - Practice and Experience*, 34:1025–1050, 2004.

[DB06]   Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. In *Proceedings of the ACM SIGPLAN Sonference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 363–382, 2006.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 522–525, 1992.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[DIS99] Claudio DeMartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[dLM07] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 50–60, New York, NY, USA, 2007. ACM Press.

[dPX+06] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.

[dSM06] Marcelo d'Amorim, Ahmed Sobeih, and Darko Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *Proceedings of International Conference on Formal Methods and Software Engineering (ICFEM)*, volume 4260, pages 549–567, 2006.

[ECDD06] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 253–264, New York, NY, USA, 2006. ACM Press.

[Ecl] Eclipse foundation. `http://www.eclipse.org/`.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131, 1996.

[Fou] Foundations of Software Engineering at Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 2001.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Procedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 40, pages 213–223, New York, NY, USA, 2005. ACM Press.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By Pierre Wolper.

[God97]   Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174–186, 1997.

[GS92]   Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[Gun92]   Carl A. Gunter. *Semantics of programming languages: structures and techniques*. MIT Press, Cambridge, MA, USA, 1992.

[Hav99]   K. Havelund. Java Pathfinder, a translator from Java to Promela. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN)*, 1999.

[HCD⁺99]   John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Proceedings of the 6th International Symposium on Static Analysis (SAS)*, pages 1–18, London, UK, 1999. Springer-Verlag.

[HDZ00]   John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher Order and Symbolic Computation*, 13(4):315–353, 2000.

[HJJJ85]   P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Towards reachability trees for high-level petri nets. pages 215–233, 1985.

[Hoa78]   C. A. R. Hoare. Communicating sequential processes. *Communications of ACM*, 21(8):666–677, 1978.

[Hol97]   G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[ID96]   C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996.

[Ios01]   Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, page 254, Washington, DC, USA, 2001. IEEE Computer Society.

[Ios02]   Radu Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of the SPIN Workshop on Software Model Checking (SPIN)*, volume 2318 of *LNCS*, pages 22–41, July 2002.

[J-S]   J-Sim. `http://www.j-sim.org/`.

[JEK⁺90]   J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[JGS93]   Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[jik]   Jikes RVM webpage. `http://jikesrvm.org/`.

[jni]   Java Native Interface: Programmer's Guide and Specification. Online book. `http://java.sun.com/docs/books/jni/`.

[jpf]   Java PathFinder webpage. `http://javapathfinder.sourceforge.net`.

[JSS00]   Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 730–733, 2000.

[KLR02]   Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 17–32, 2002.

[KP89]   Shmuel Katz and Doron Peled. An efficient verification method for parallel and distributed programs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 489–507, London, UK, 1989. Springer-Verlag.

[KPV03]   Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, April 2003.

[LH04]   Ondrej Lhotak and Laurie Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation (PLDI)*, pages 158–169, New York, NY, USA, 2004. ACM Press.

[LV01]   Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Proceedings of the international SPIN workshop on Model checking of software (SPIN)*, pages 80–102, Toronto, Canada, 2001.

[McM92]   Kenneth L. McMillan. *Symbolic Model Checking*. Ph.D., Carnegie Mellon University, Pittsburgh, PA, May 1992.

[McM93]   Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM00]   Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[MD05]   Madanlal Musuvathi and David L. Dill. An incremental heap canonicalization algorithm. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN)*, pages 28–42, 2005.

[ME04]   M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, pages 155–168, 2004.

[MK01]   Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 22–31, 2001.

[MPC+02]   Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, December 2002.

[MS01]   Alexandre Mota and Augusto Sampaio. Model checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40(1):59–96, 2001.

[MVP]   Peter C. Mehlitz, Willem Visser, and John Penix. The JPF runtime verification system. Online manual. `http://javapathfinder.sourceforge.net/JPF.pdf`.

[NIS02]   National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.

[OJP99]   A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Software - Practice and Experience*, 29(2):167–193, 1999.

[oLS05]   Bureau of Labor Statistics. Current population survey (cps). computer ownership/internet usage supplement. 2005, 2005. `http://www.census.gov/population/socdemo/computer`.

[PE05]   Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 504–527, Glasgow, Scotland, July 2005.

[PR99]   Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100. IEEE Computer Society Press, 1999.

[PSSD00]   D. Y.W. Park, U. Stern, J. U. Skakkebæk, and D. L. Dill. Java model checking. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2000.

[PVE+00]   John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 488–497, New York, NY, USA, 2000. ACM Press.

[Qad]   Shaz Qadeer. Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. 2004.

[RDH03]   Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proceedings of the European Software Engineering Conference and SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 267–276, 2003.

[RDHI03]   Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.

[Ros94]   A. W. Roscoe. *A classical mind: essays in honour of C. A. R. Hoare*, chapter Model Checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[Rug04]   Radu Rugina. Shape analysis quantitative shape analysis. In *Proceedings of the Static Analysis Symposium (SAS)*, pages 228–245, 2004.

[SBN+97]   Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SLA02]   David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In *Proceedings of the XP/Agile Universe Conference*, pages 131–143, 2002.

[SM03]   Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[SMA05]   Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, September 2005.

[SSM05]   Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.

[Sun]   Sun Microsystems. *jstat: Java Virtual Machine Statistics Monitoring Tool*. `http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html`.

[SVH04]   Ahmed Sobeih, Mahesh Viswanathan, and Jennifer C. Hou. Incorporating bounded model checking in network simulation: Theory, implementation and evaluation. Technical Report UIUCDCS-R-2004-2466, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, July 2004.

[SVMH05]   Ahmed Sobeih, Mahesh Viswanathan, Darko Marinov, and Jennifer C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*, pages 235–250, 2005.

[Tar55]   Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[TSH05]   Hung-Ying Tyan, Ahmed Sobeih, and Jennifer C. Hou. Towards composable and extensible network simulation. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2005.

[Tya02]   H.-Y. Tyan. *Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*. Ph.D., Department of Electrical Engineering, The Ohio State University, 2002.

[Val91]   Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the International Workshop on Computer Aided Verification*, pages 156–165, London, UK, 1991. Springer-Verlag.

[VCST05]   Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online test-ing with model programs. In *Proceedings of the European Software Engineering Confer-ence and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 273–282, New York, NY, 2005. ACM Press.

[VHB⁺03]   Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[VHBP00]   Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the International Conference on Automated Software En-gineering (ASE)*, volume 00, pages 3–12, Washington, DC, USA, 2000. IEEE Computer Society.

[VPK04]   Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107, 2004.

[VPP00]   William Visser, SeungJoon Park, and John Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the Workshop on Formal Methods in Software Practice (FMSP)*, pages 3–182, 2000.

[VPP05]   Willem Visser, Corina S. Pasareanu, and Radek Pelanek. Test input generation for red-black trees using abstraction. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 414–417, 2005.

[VPP06]   Willem Visser, Corina S. Pasareanu, and Radek Pelanek. Test input generation for Java containers using state matching. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.

[Wei81]   Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ASE)*, pages 439–449, 1981.

[Win93]   Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[WL04]   John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[WS06]   Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engeneering*, 32(2):93–110, 2006.

[XMN04]   Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redun-dant object-oriented unit tests. In *Proceedings of the IEEE/ACM International Con-ference on Automated Software Engineering (ASE)*, pages 196–205, September 2004.

[XMSN05]   Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 365–381, April 2005.

[YRS04]   Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.

[YTEM04]   Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, pages 273–288, 2004.

[ZMS$^+$07]   Yuanyuan Zhou, Darko Marinov, William Sanders, Craig Zilles, Marcelo d'Amorim, Steven Lauterburg, Ryan M. Lefever, and Joe Tucek. Delta execution for software reliability. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2007.