# Optimized Execution of Deterministic Blocks in Java PathFinder

Marcelo d'Amorim, Ahmed Sobeih, and Darko Marinov

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana IL, 61801 USA
{damorim, sobeih, marinov}@cs.uiuc.edu

**Abstract.** Java PathFinder (JPF) is an explicit-state model checker for Java programs. It explores all executions that a given program can have due to different thread interleavings and nondeterministic choices. JPF implements a backtracking Java Virtual Machine (JVM) that executes Java bytecodes using a *special representation* of JVM states. This special representation enables JPF to quickly store, restore, and compare states; it is crucial for making the overall state exploration efficient. However, this special representation creates overhead for each execution, even execution of *deterministic blocks* that have no thread interleavings or nondeterministic choices.

We propose *mixed execution*, a technique that reduces execution time of deterministic blocks in JPF. JPF is written in Java as a special JVM that runs on top of a regular, host JVM. Mixed execution works by translating the state between the special JPF representation and the host JVM representation. We also present *lazy translation*, an optimization that speeds up mixed execution by translating only the parts of the state that a specific execution dynamically depends on. We evaluate mixed execution on six programs that use JPF for generating tests for data structures and on one case study for verifying a network protocol. The results show that mixed execution can improve the overall time for state exploration up to 36.98%, while improving the execution time of deterministic blocks up to 69.15%. Although we present mixed execution in the context of JPF and Java, it generalizes to any model checker that uses a special state representation.

## 1 Introduction

Software model checking [3, 6, 12, 27, 19, 32] is a promising approach for increasing the reliability of programs. The goal of model checking is to explore the program's state space to find property violations or confirm absence of violations. While "state-space explosion" is the key issue in model checking, time efficiency is also an important problem. Several recent model checking tools—including AsmLT [12], BogorVM [27], JPF [19], and SpecExplorer [32]—make the trade-off to speed up the overall state exploration by slowing down a straight-line execution. This work focuses on speeding up the straight-line execution.

We present our approach in the context of the Java PathFinder (JPF) [33, 19], an explicit-state model checker for Java programs. JPF takes as input a Java program and an optional bound on the length of program execution. JPF explores all executions (up

Z. Liu and J. He (Eds.): ICFEM 2006, LNCS 4260, pp. 549–567, 2006.

to the given bound) that the program can have due to different thread interleavings and nondeterministic choices. JPF can generate as output those executions that violate a given (temporal) property, for example violate an assertion or lead to a deadlock. JPF can also generate as output test inputs for the given program [34, 35].

JPF is implemented in Java as a special Java Virtual Machine (JVM) that runs on top of the host JVM. The main difference between JPF and a regular JVM is that JPF can (quickly) backtrack the program execution by restoring any state previously encountered during the execution. Backtracking allows exploration of different executions from the same state. To achieve fast backtracking, JPF uses a *special representation* of states and executes program bytecodes by modifying this representation. The special state representation makes the overall exploration of *all different executions efficient*, although it makes *each single execution inefficient* compared to a regular JVM. An alternative to using special state representation is using the native state representation of the host JVM throughout model checking; however, while native representation makes each single execution efficient, it can slow down the overall exploration.

We propose *mixed execution*, a technique that can reduce execution time in JPF. The main idea of mixed execution is to execute *some* parts of the program not on JPF but directly on the host JVM. With mixed execution, JPF still as usual executes the other parts of the program and stores, restores, and compares the states. Mixed execution executes on the host JVM only *deterministic blocks*, i.e., parts of the execution that have no thread interleavings or nondeterministic choices. To achieve this, mixed execution translates the state from JPF to JVM at the beginning of a block and from JVM to JPF at the end of a 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. Although we present mixed execution in the context of JPF, our main idea—executing parts of model checking on different state representations—generalizes to all other model checkers—including AsmLT [12], BogorVM [27], and SpecExplorer [32]—that use some special state representation; these checkers do not need to be for Java or even based on virtual machines.

We have implemented 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 [19]:

> 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.

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 the execution* from the state-tracked JPF into the non-state tracked host JVM *even for components that are property-relevant*. Indeed, mixed execution executes on the host JVM some program code that can modify the program state and thus affect a property, for example assertion violation. For example, we use our technique in the execution of property-relevant fragments during the model checking of a network protocol. In contrast, the previous use of MJI in JPF did not execute such program code on JVM and did not translate the state between JPF and JVM representations.

We also present *lazy translation*, an optimization that speeds up mixed execution by translating only the parts of the state that an execution dynamically depends on. The basic, *eager* mixed execution always translates from JPF to JVM the entire state reachable from a set of roots at the beginning of a deterministic block. (Note that even this state can be a tiny part of the entire JVM state.) 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. We have implemented lazy translation by providing an instrumentation for the classes executed on the host JVM.

We evaluate mixed execution and lazy translation on six subject programs that use JPF to generate tests for data structures. The experimental results show that mixed execution can improve the overall time for state exploration in JPF up to 36.98%, while improving the time for execution of deterministic blocks up to 69.15%. Additionally, lazy translation can improve the eager mixed execution up to 25.02%. We also evaluate mixed execution on a case study that uses JPF to find a bug in a fairly complex routing protocol, AODV [25]. Note that mixed execution only reduces the execution time for deterministic blocks and thus the overall exploration time; mixed execution does not affect the order of exploration, the number of explored states, or any other aspect of the state exploration. The techniques that improve the latter aspects are orthogonal to mixed execution, which can be used to further improve them.

## 2 Example

We next present an example that illustrates how mixed execution can speed up JPF's state exploration. Figure 1 shows the example code that was previously used in several studies on JPF [34, 35, 36]. The code explores the state space of the `java.util.TreeMap` class from the standard Java libraries. This class implements the map interface using red-black trees. The basic operations on the map are `put` (which adds a given key-value pair; the example sets all values to `null`), `remove` (which removes the key-value pair for a given key), and `get` (which gets the value for a given key). The code represents a driver that explores all sequences of `put`, `remove`, and `get` operations up to the given bounds `M` (for the sequence length) and `N` (for the range of input values). JPF's library method `Verify.random(int n)` nondeterministically returns a number between zero and the given bound `n`. JPF's library methods `beginAtomic` and `endAtomic` mark an *atomic* block; these (manually added) annotations instruct JPF to ignore thread interleavings within a given block.

Figure 1 shows relevant fields and methods of the class `TreeMap`. Objects of the `Entry` class represent the nodes of red-black trees. Each node has a key-value pair, a color (red or black), and pointers to the parent node and the left and right children. Executions of the `put`, `remove`, and `get` methods manipulate the tree (passed as the implicit `this` argument). The goal of the driver is to explore different trees that can

```
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
    // initialize N method arguments
    Integer[] elems = new Integer[N];
    for (int i = 0; i < N; i++) elems[i] = new Integer(i);
    // create an empty tree, the root object for exploration
    TreeMap t = new TreeMap();
    // explore method sequences up to length M
    for (int i = 0; i < M; i++) {
        Verify.beginAtomic();
        switch (Verify.random(2)) {
        case 0: t.put(elems[Verify.random(N-1)], null); break;
        case 1: t.remove(elems[Verify.random(N-1)]);  break;
        case 2: t.get(elems[Verify.random(N-1)]); break;
        }
        Verify.endAtomic();
        Verify.ignoreIf(storeIfNotAlreadyStored(t));
    }
}

public class TreeMap {
    Entry root;
    int size;
    static class Entry {
        Object key;
        Object value;
        boolean color;
        Entry left;
        Entry right;
        Entry parent; ...
    }
    public Object put(Object key, Object value) { ... }
    public Object remove(Object key) { ... }
    public Object get(Object key) { ... } ...
}
```

**Fig. 1.** Driver for bounded-exhaustive exploration and parts of TreeMap code

arise during the executions. JPF in general considers the entire state when comparing different executions, but the driver uses *abstract matching* [35,36,37] to compare only the state of the tree, namely the state of all objects reachable from the root `t`. If the state has been already visited, the JPF's library method `Verify.ignoreIf` instructs JPF to backtrack the execution.

As already mentioned, JPF uses a special representation of the JVM state to efficiently store, restore, and compare states. Without mixed execution, JPF executes `put`, `remove`, and `get` methods on the special representation, which slows down every field read and write. Note, however, that JPF needs the state of the tree only at the beginning and at the end of these methods; in other words, each method can execute atomically. Mixed execution therefore executes these three methods on the host JVM:

- At the beginning of each method execution, mixed execution translates the objects reachable from the method parameters (including the tree reachable from `this`) from the JPF representation into the host JVM representation. (Lazy translation does not translate all objects at the beginning but only on demand during the execution.)
- Mixed execution then invokes the method on the translated state in the host JVM. The method execution can then modify this state.

– At the end of each method execution, mixed execution translates the state back from the host JVM representation into the JPF representation. JPF then compares whether it has already explored the resulting state, appropriately backtracks the execution (restores the state), and the process continues.

The speedup (or slowdown) that mixed execution achieves depends on the size of the state and the length of the method execution. The smaller the state is, the less mixed execution has to copy between the JPF and JVM representations. (Lazy translation further reduces this cost such that it does not depend on the size of the state at the beginning of the method but on the size of the state that the execution accesses.) Also, the longer the execution is, the more mixed execution saves by executing on JVM rather than on JPF.

In our running example with `TreeMap`, the results depend on the value for the bounds `M` and `N`. We set `M = N` in all experiments, and the value ranges from 6 to 10, as done in the previous studies with abstract matching [35, 36, 37]. For these bounds, JPF with mixed execution (and lazy translation) takes from 9.44% to 36.98% less time for overall state exploration than JPF without mixed execution. Considering only the executions of `put`, `remove`, and `get` methods, mixed execution provides from 43.15% to 54.95% speedup. Besides the executions of these methods, the overall state exploration includes state comparison, backtracking, and other JPF operations. Mixed execution only reduces the method execution time, while the cost of the rest of state exploration remains the same.

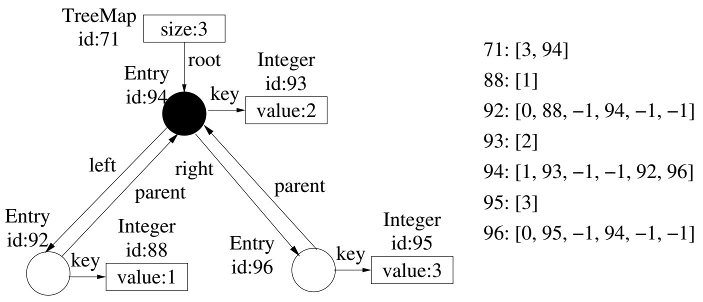
## 3 Background

We briefly review the parts of JPF relevant for mixed execution. More details on JPF can be found elsewhere [19, 33]. We first describe how JPF represents state. More specifically, we focus on how JPF represents the heap. While JPF also represents stack, thread information, class information, and all other parts of a JVM state, mixed execution directly manipulates only the heap. We then describe the Model Java Interface (MJI), an existing mechanism in JPF for accessing the JPF state from the host JVM. Mixed execution uses MJI to translate the heap between the JPF and JVM representations.

### 3.1 Heap Representation

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 pointer to another object (which can hold the special value `null`).

Recall that JPF is implemented in Java. 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 is an array of integer arrays. Figure 2 shows an example red-black tree represented in JVM (as an object graph) and in JPF (as an array of integer arrays); this example `TreeMap` object can result from the sequence `TreeMap t = new TreeMap(); t.put(new Integer(2), null); t.put(new Integer(1), null); t.put(new Integer(3), null)`. Figure 2 shows for each object its type,



```
71: [3, 94]
88: [1]
92: [0, 88, −1, 94, −1, −1]
93: [2]
94: [1, 93, −1, −1, 92, 96]
95: [3]
96: [0, 95, −1, 94, −1, −1]
```

**Fig. 2.** An example TreeMap as an object graph and in the JPF heap representation

integer identifier, and the values of primitive fields (with the full and empty circles representing the color of the `Entry` objects). The pointer fields not shown in the graph have the value `null`, represented as -1 in JPF.

### 3.2 Model Java Interface

Model Java Interface (MJI) is a JPF mechanism that allows parts of JPF execution to be delegated from the JPF into the host JVM. MJI is analogous to the Java Native Interface (JNI) [2] that allows parts of JVM execution to be delegated from the JVM into the native code, written in say the C language. MJI, like JNI, splits executions at the method granularity; namely, each method can be marked to be executed either in 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. MJI, like JNI, is used to implement functionality that either requires higher performance or is not available at the target level (e.g., reflection [11] in Java). Specifically, JPF uses MJI to implement several classes and methods from the `java.io` and `java.lang` packages. These existing methods do not modify the heap; they either only affect the IO or only return primitive values or new objects. In contrast, our mixed execution leverages MJI to execute code that can and does modify the heap. Also, mixed execution does not operate on the JPF representation of state; instead, mixed execution translates the state between the JPF representation and the host JVM representation.

## 4 Technique

We next present mixed execution in more detail. Like MJI and JNI (Section 3), mixed execution operates at the method granularity: the user can mark each method to be executed either in JPF or in the host JVM. We first present how mixed execution invokes the methods to be executed on the host JVM. We then present the basic version of mixed execution that eagerly translates the state between JPF and JVM at the boundaries of a

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

**Fig. 3.** Pseudo-code of the method invocation for the host JVM execution

method call. We finally present lazy translation, an optimization that translates only the parts that the execution actually needs.

## 4.1 Overview

Figure 3 shows how mixed execution invokes methods for host execution. Whenever the program is about to execute a method, mixed execution checks whether the method is marked to be executed in the host JVM. If so, mixed execution translates the state from JPF to JVM, executes the method, and then translates the state back from JVM to JPF. Note that mixed execution handles both cases when the method returns normally and when the method throws an exception; mixed execution catches the (JVM) exceptions and translates them accordingly (into the JPF exceptions), together with the rest of the post-state.

Mixed execution assumes that the methods marked for execution in the host JVM are deterministic, i.e., are not affected by any interleaving of threads and have no nondeterministic choices. (This is always the case when JPF is used to explore method sequences as shown in Section 5.1; the code is single-threaded and there are no `Verify.random` calls in the methods.) Each method takes several arguments (one of which is the implicit `this` argument for instance methods). Some of the arguments may be pointers to objects, and a method execution can access or modify a field of any object reachable from these pointers. The arguments thus represent the 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, the roots should also include all static fields.)

## 4.2 Eager Translation

Figure 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 envi-

```
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 the new object with all fields translated
    return o;
}
// 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;
}
```

**Fig. 4.** Pseudo-code of the algorithm that translates the state from JPF to JVM

ronment/state of the JPF execution, and an array of method arguments, encoded in JPF as integers (Section 3). (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 [5]. 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

```
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);
}
```

**Fig. 5.** Pseudo-code of the algorithm that translates the state from JVM to JPF

several MJI calls (on the `env` objects) to get the values of fields and to get the types of the arguments and fields.

Figure 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 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) to create new objects and set the values of fields.

```
// 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
    } ...
}
```

**Fig. 6.** Example code before and after instrumentation

## 4.3 Lazy Translation

Lazy translation is an optimization that translates between JPF and JVM only the parts of the heap that a method execution actually needs. While eager translation translates the entire heap at the beginning of the execution, lazy translation 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 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 [5, 34].

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.

## 5 Experiments

We next discuss the experiments used to evaluate mixed execution. We have implemented mixed execution by modifying the JPF code [19] to include the algorithms from figures 3, 4, and 5. We have also implemented a prototype tool that automates instrumentation for lazy translation as shown in Figure 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 non-deterministic choices (`Verify.random` calls).

We conducted all the experiments on a dual-processor Intel Xeon 2.8 GHz machine running Linux version 2.6.15 with 2 GB memory. We used Sun's 1.4.2_06-b03 JVM, allocating 1.5 GB for the maximum heap size. 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 [33], the use of MD5 hashing function [19], and the exact state comparison with respect to isomorphism [35, 36].

### 5.1 Data Structures

We evaluate mixed execution on the six data structures listed in Figure 7. We take the subjects from previous studies on model checking and testing:

– `UBStack` is an implementation of a stack bounded in size, storing integer objects without repetition [30, 37, 8, 23].
– `DisjSet` is an implementation of a union-find data structure implementing disjoint sets [37].
– `Trie` implements a dictionary, i.e., it stores a collection of strings sorted lexicographically [38].
– `Vector`, `LinkedList`, and `TreeMap` are from the Java 1.4 Collection Framework.

Our state exploration considers the methods that add, remove, and search for elements in each data structure, as listed in Figure 7.

Each experiment uses an execution driver similar to that in Figure 1. By default, we use mixed execution with lazy translation. Figure 8 tabulates the results. We set

| subject | methods explored |
|---|---|
| UBStack | push, pop |
| DisjSet | union, find |
| Trie | add, is_word, is_proper_prefix |
| Vector | addElement, removeElement, elementAt |
| LinkedList | add, removeLast, contains |
| TreeMap | put, remove, get |

**Fig. 7.** Subjects used in the experiments

| | | | # bytecodes | | total time | | | method exec. only | | |
|---|---|---|---|---|---|---|---|---|---|---|
| subject | bound | # states | JPF | mixed | JPF | mixed | speedup | JPF | mixed | speedup |
| | | | | | [ms] | [ms] | [%] | [ms] | [ms] | [%] |
| UBStack | 5 | 929 | 181217 | 19677 | 2318 | 2137 | 7.81 | 490 | 276 | 43.67 |
| UBStack | 6 | 5776 | 1561823 | 132475 | 5605 | 4367 | 22.09 | 1836 | 726 | 60.46 |
| UBStack | 7 | 41094 | 14940706 | 1038230 | 31602 | 20889 | 33.90 | 14301 | 4412 | 69.15 |
| DisjSet | 5 | 624 | 207261 | 21507 | 2546 | 2500 | 1.81 | 187 | 233 | -24.60 |
| DisjSet | 6 | 4653 | 2067901 | 161408 | 9602 | 8902 | 7.29 | 1146 | 789 | 31.15 |
| DisjSet | 7 | 47480 | 27152409 | 1874435 | 92054 | 82169 | 10.74 | 14133 | 8194 | 42.02 |
| Trie | 5 | 129 | 120869 | 4839 | 1686 | 1636 | 2.97 | 262 | 221 | 15.65 |
| Trie | 6 | 257 | 293899 | 10855 | 2068 | 1966 | 4.93 | 419 | 287 | 31.50 |
| Trie | 7 | 513 | 690129 | 24359 | 2804 | 2572 | 8.27 | 752 | 460 | 38.83 |
| Trie | 8 | 1025 | 1679127 | 54311 | 4834 | 4149 | 14.17 | 1440 | 847 | 41.18 |
| Trie | 9 | 2049 | 4018501 | 120103 | 9329 | 7730 | 17.14 | 2929 | 1446 | 50.63 |
| Trie | 10 | 4097 | 9190465 | 263549 | 18946 | 15599 | 17.67 | 6547 | 3064 | 53.20 |
| Vector | 5 | 7057 | 892349 | 120244 | 4513 | 4074 | 9.73 | 1001 | 522 | 47.85 |
| Vector | 6 | 91706 | 13596654 | 1605126 | 38360 | 30534 | 20.40 | 11504 | 4462 | 61.21 |
| Vector | 7 | 1466919 | 247371240 | 26241690 | 1276992 | 1124545 | 11.94 | 206508 | 74046 | 64.14 |
| LinkedList | 5 | 5471 | 302914 | 105134 | 4390 | 4256 | 3.05 | 914 | 808 | 11.60 |
| LinkedList | 6 | 74652 | 4218361 | 1446823 | 35109 | 33453 | 4.72 | 10128 | 9367 | 7.51 |
| LinkedList | 7 | 1235317 | 70962644 | 24157788 | 578847 | 553095 | 4.45 | 175267 | 151016 | 13.84 |
| TreeMap | 5 | 187 | 92740 | 7586 | 1841 | 1735 | 5.76 | 364 | 201 | 44.78 |
| TreeMap | 6 | 534 | 361600 | 25864 | 2532 | 2293 | 9.44 | 761 | 410 | 46.12 |
| TreeMap | 7 | 1480 | 1223256 | 79470 | 4294 | 3490 | 18.72 | 1738 | 988 | 43.15 |
| TreeMap | 8 | 4552 | 4629574 | 277476 | 10489 | 7556 | 27.96 | 5718 | 2805 | 50.94 |
| TreeMap | 9 | 13816 | 16681289 | 952976 | 32254 | 20897 | 35.21 | 20096 | 9424 | 53.11 |
| TreeMap | 10 | 39344 | 54581750 | 3008954 | 98633 | 62162 | 36.98 | 66336 | 29887 | 54.95 |

**Fig. 8.** Comparison of JPF without and with mixed execution

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.

The results show that mixed execution can reduce the overall state exploration time up to 36.98%, while reducing the method execution time up to 69.15%. 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 small bounds, mixed execution slows

| | | | total time | | | method exec. only | | |
|---|---|---|---|---|---|---|---|---|
| name | bound | # states | eager | lazy | speedup | eager | lazy | speedup |
| | | | [ms] | [ms] | [%] | [ms] | [ms] | [%] |
| Trie | 5 | 129 | 1785 | 1748 | 2.07 | 234 | 202 | 13.68 |
| Trie | 6 | 257 | 2118 | 2052 | 3.12 | 393 | 295 | 24.94 |
| Trie | 7 | 513 | 2842 | 2650 | 6.76 | 635 | 465 | 26.77 |
| Trie | 8 | 1025 | 4958 | 4574 | 7.75 | 1660 | 833 | 49.82 |
| Trie | 9 | 2049 | 10150 | 7911 | 22.06 | 3595 | 1446 | 59.78 |
| Trie | 10 | 4097 | 20678 | 15730 | 23.93 | 8217 | 3018 | 63.27 |
| TreeMap | 5 | 187 | 1842 | 1820 | 1.19 | 305 | 218 | 28.52 |
| TreeMap | 6 | 534 | 2685 | 2651 | 1.27 | 571 | 403 | 29.42 |
| TreeMap | 7 | 1480 | 3901 | 3498 | 10.33 | 1338 | 962 | 28.10 |
| TreeMap | 8 | 4552 | 9089 | 7548 | 16.95 | 4308 | 2864 | 33.52 |
| TreeMap | 9 | 13816 | 27595 | 21014 | 23.85 | 15235 | 9425 | 38.14 |
| TreeMap | 10 | 39344 | 83744 | 62789 | 25.02 | 49212 | 29600 | 39.85 |

**Fig. 9.** Comparison of eager and lazy translations

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.

All above experiments with mixed execution use lazy translation. Figure 9 shows the benefit of this optimization. For two subjects and several sizes, we tabulate the overall execution time for state exploration and the time for execution of methods marked for mixed execution. Compared to eager translation, lazy translation reduces the overall time up to 25.02%, while reducing the method execution time up to 63.27%. Note again that the longer the execution gets, the more benefit lazy translation provides.

### 5.2 The AODV Case Study

We next present the evaluation of mixed execution on Ad-Hoc On-Demand Distance Vector (AODV) routing [26], a widely used network protocol for wireless multihop ad hoc networks. We consider an implementation of AODV based on the AODV Draft (version 11) [25] and implemented in J-Sim [1,31], a component-based 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 [28,29].

We first give an overview of AODV and its *loop-free* safety property. We then explain the details of the driver for AODV and an error that we injected in the AODV code. We finally present the improvements obtained by using mixed execution to find the error.

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. Hence, the route between two nodes may consist of multiple wireless hops through other nodes; this is called *multihop routing*.

In AODV, each node $n$ in the ad hoc network maintains a routing table. A routing table entry (RTE) at node $n$ to a destination node $d$ contains, among other fields: a next hop address $nexthop_{n,d}$ (the address of the node to which $n$ forwards packets destined for $d$), a hop count $hops_{n,d}$ (the number of hops needed to reach $d$ from $n$), and a destination sequence number $seqno_{n,d}$ (a measure of the freshness of the route information). Each RTE is associated with a lifetime. Periodically, a route timeout event is triggered invalidating (but not deleting) all the RTEs that have not been used (e.g., to send or forward packets to the destination) for a time interval that is greater than the lifetime. Invalidating a RTE involves incrementing $seqno_{n,d}$ and setting $hops_{n,d}$ to $\infty$.

Each node $n$ also maintains two monotonically increasing counters: a node sequence number $seqno_n$ and a broadcast ID $bid_n$. When node $n$ requires a route to a destination $d$ to which $n$ does not already have a valid RTE, $n$ creates an invalid RTE to $d$ with $hops_{n,d}$ set to $\infty$. Node $n$ then *broadcasts* a route request (RREQ) packet with the fields $\langle n, seqno_n, bid_n, d, seqno_{n,d}, hopCount_q \rangle$ and increments $bid_n$. The $hopCount_q$ field is initialized to 1. The pair $\langle n, bid_n \rangle$ uniquely identifies a RREQ packet. Each node $m$, receiving the RREQ packet from node $n$, keeps the pair $\langle n, bid_n \rangle$ in a broadcast ID cache so that $m$ can later check if it has already received a RREQ with the same source address and broadcast ID. If so, the incoming RREQ packet is discarded. If not, $m$ either satisfies the RREQ by *unicasting* a route reply (RREP) packet back to $n$ if it has a fresh enough route to $d$ (or it is $d$ itself) or rebroadcasts the RREQ to its own neighbors after incrementing the $hopCount_q$ field if it does not have a fresh enough route to $d$ (nor is it $d$). An intermediate node $m$ determines whether it has a fresh enough route to $d$ by comparing the destination sequence number $seqno_{m,d}$ in its own RTE with the $seqno_{n,d}$ field in the RREQ packet. Each intermediate node also records a reverse route to the requesting node $n$; this reverse route can be used to send/forward route replies to $n$. The requesting node's sequence number $seqno_n$ is used to maintain the freshness of this reverse route. 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.

**Overview of AODV.** A RREP packet, which is sent by an intermediate node $m$, contains the following fields $\langle hopCount_p, d, seqno_{m,d}, n \rangle$. The $hopCount_p$ field is initialized to $1 + hops_{m,d}$. If it is the destination $d$ that sends the RREP packet, it first increments $seqno_d$ and then sends a RREP packet containing the following fields $\langle 1, d, seqno_d, n \rangle$. The unicast RREP travels back to the requesting node $n$ via the reverse route. Each intermediate node along the reverse route sets up a forward pointer to the node from which the RREP came, thus establishing a forward route to the destination $d$, increments the $hopCount_p$ field and forwards the RREP packet to the next hop towards $n$.

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-free* property. Intuitively, a node must not exist at two points on 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,

| AODV | | | # bytecodes | | total time | | | method exec. only | | |
|---|---|---|---|---|---|---|---|---|---|---|
| # nodes | path len. | # states | JPF | mixed | JPF | mixed | speedup | JPF | mixed | speedup |
| | | | | | [ms] | [ms] | [%] | [ms] | [ms] | [%] |
| 8 | 8 | 5806 | 24571290 | 17425293 | 61347 | 54384 | 11.35 | 9107 | 3457 | 62.04 |
| 9 | 9 | 7960 | 37106325 | 26683825 | 92266 | 82231 | 10.88 | 13520 | 4892 | 63.82 |
| 10 | 10 | 10585 | 54077303 | 39272619 | 161578 | 110132 | 31.84 | 19495 | 6578 | 66.26 |

**Fig. 10.** Model checking AODV without and with mixed execution

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 [4, 22]:

$$seqno_{n,d} < seqno_{m,d} \vee (seqno_{n,d} = seqno_{m,d} \wedge hops_{n,d} > hops_{m,d})$$

**Test driver.** We wrote a test driver for the J-Sim implementation of AODV. The driver produces an environment that executes all sequences of protocol events up to a configurable bound. The driver considers these events [29]:

- Initiation of a route request to a destination $d$: This event is enabled if the node does not have a valid RTE to the destination $d$. The event is handled by broadcasting a RREQ.
- Restart of the AODV process at node $n$: This event may take place because of a node reboot. The event is always enabled and is handled by reinitializing the state of the AODV process at node $n$.
- Broadcast ID timeout at node $n$: This event is enabled if there is at least one entry in the broadcast ID cache of node $n$. The event is handled by deleting this entry.
- Timeout of the route to destination $d$ at node $n$: This event is enabled if $n$ has a valid RTE to $d$. The event is handled by invalidating this RTE.
- Delivering an AODV packet to node $n$: This event is enabled if the network contains at least one AODV packet such that $n$ is the destination (or the next hop towards the destination) of the packet and $n$ is one of the neighbors of the source of the packet. The event is handled by removing this packet from the network and forwarding it to node $n$ in order to be processed according to the AODV implementation.
- Loss of an AODV packet destined for node $n$: This event is enabled if the network contains at least one AODV packet that is destined for node $n$. The event is handled by removing this packet from the network.

Since JPF could not execute the code for the entire J-Sim simulator and the AODV protocol, we created a simplified version of the networking layer used by AODV. This version does not have the full generality of the J-Sim simulator but provides the functionality needed to run AODV.

**Finding error.** We consider an initial state of an ad hoc network consisting of $K$ nodes: $n_0$, $n_1$, $\ldots$, $n_{K-1}$ (where $n_{K-1}$ is the only destination node) arranged in a chain topology where each node is a neighbor of both the node to its left and the node to its right (if they exist). In the initial state, nodes $n_i$ for all $0 \leq i \leq K-2$ have valid routing table entries to the destination $n_{K-1}$. We manually injected an error as follows:

a RTE is deleted (instead of invalidated) when a route timeout event occurs. Consider the case of $K = 3$. A routing loop may occur because if $nexthop_{0,2} = 1$ and a route timeout event takes place at $n_1$, if $n_1$ is later offered a route to $n_2$ by $n_0$, this route will be accepted because $seqno_{0,2} > seqno_{1,2}$. The case of $K > 3$ is similar. The interested reader can find a detailed explanation of this injected error elsewhere [28]. We instruct JPF to stop the exploration as soon as it finds this error.

**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 for host execution the data structures (such as vectors) that AODV uses to represent protocol data (including routing tables and packet queues). We then used profiling to find that AODV spends a lot of execution time in the methods of the J-Sim library class `Port` that handles sending and receiving of packets between network nodes [31], so we also marked those methods for host execution. Figure 10 shows the improvements obtained with mixed execution on AODV. We tabulate, for a range of number of nodes and length of the event path, the overall state-space exploration time and the method execution time. Mixed execution improves the overall exploration time from 10.88% to 31.84%, and the method execution time from 62.04% to 66.26%.

## 6 Related Work

Traditional model checkers such as SPIN [15], SMV [18], or Murphi [9] have been extensively used in formal reasoning of both hardware and software systems. These tools analyze the models written in the special modeling languages. To analyze a system, the user thus needs either to manually write a model of the system in a language understood by the tools [4] or to automatically translate an implementation of the system from a programming language (e.g., Java) into the modeling language of the tools [24,14,7,10]. Our work considers model checkers that directly analyze the systems written in a programming language.

Verisoft [13] was the first model checker to directly analyze the implementation code, specifically code written in the C language. Several recent model checkers such as CMC [22], BogorVM [27], or JPF [33] also focus on analyzing the actual code written in a programming language (C or Java). For example, CMC has been used to model check Linux implementations of networking code (e.g., AODV and TCP) and file systems [22, 21, 39]. We have also developed a model checker [29] tailored for the J-Sim network simulator [1] and used it to find errors in the J-Sim implementation of AODV [29]. The model checker extends J-Sim with the capability to explore the state space created by a network protocol, whose simulation code is written in Java. The model checker operates on the concrete memory state and clones/copies large portions of the state for each transition. Our current work targets model checkers that operate on a special representation of state such as AsmLT, BogorVM, JPF, or SpecExplorer.

Handling state is a central issue in explicit-state model checkers [15,17,16,20]. Work in this area focuses on efficient implementation of state operations, including updating, storing, restoring, and comparing states. For example, JPF implements techniques such as efficient encoding of Java program state and symmetry reductions to help reduce the state-space size [17]. As another example, Musuvathi and Dill recently proposed an algorithm for incremental heap canonicalization [20], which speeds up the hashing of states and thus state comparisons. While these techniques focus on *speeding up the operations* on state (or the overall state-space exploration), we propose mixed execution that focuses on *speeding up the executions* that operate on the state that can be translated between the special (JPF) and the host (JVM) representation. Our technique is thus orthogonal to the techniques for state operations and can be combined with them to achieve even higher speed ups.

Our evaluation of mixed execution uses data-structure subjects and the AODV case study. The data-structure subjects have been used in other projects on testing and model checking [30, 37, 8, 38, 23], including in the context of JPF [34, 35, 36]. The most recent work in the context of JPF [35, 36] proposes test-input generation techniques that depend on the *abstract state matching* to avoid the generation of redundant tests. Our experiments rely on that work because our drivers match the state of the data structure (reachable from a root) and not the entire heap. As the results show, however, mixed execution still achieves significant improvements even when used with abstract matching. Finally, the AODV case study presented in this paper is, to the best of our knowledge, one of the largest case studies that have been model checked using JPF.

## 7 Conclusions

We have presented mixed execution, a technique that reduces the execution time of deterministic blocks in Java PathFinder (JPF). JPF is a special JVM that runs on top of a regular, host JVM; mixed execution translates the state between the special JPF representation and the host JVM representation to enable faster execution of Java bytecodes. We have also presented lazy translation, an optimization that speeds up mixed execution by translating only the parts of the state that an execution dynamically depends on. Our evaluation shows that mixed execution can significantly improve the time for execution of deterministic blocks and thus the overall time for state-space exploration.

Mixed execution points out the importance of studying the trade-offs used in state-space explorations for model checking and testing. We plan to further investigate these trade-offs, focusing on the differences between stateful and stateless search (i.e., between backtracking and re-execution). We also plan to consider the use of memoization and incremental computation in speeding up re-execution. We believe that the straight-line execution in model checkers can be further improved, building on the ideas of mixed execution.

## Acknowledgments

We thank Willem Visser for repeatedly and promptly helping us with JPF, Corina Pasareanu and Mahesh Viswanathan for discussing mixed execution with us, and Steven Lauterburg and Nikolai Tillmann for their comments on a previous draft of this paper. This work was partially supported by CAPES fellowship under grant #15021917. We also acknowledge support from Microsoft Research.

## References

1. J-Sim. `http://www.j-sim.org/`.
2. Java Native Interface: Programmer's Guide and Specification. Online book. `http://java.sun.com/docs/books/jni/`.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proc. of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, 2001.
4. 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.
5. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 168–176, 2004.
7. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering*, pages 439–448, 2000.
8. C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.
9. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design (IEEE ICCD)*, pages 522–525, 1992.
10. A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *Proc. of CAV'04*, July 2004.
11. D. Flanagan. *Java In A Nutshell*. O'Reilly, 1997.
12. Foundations of Software Engineering at Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.
13. P. Godefroid. Model checking for programming languages using Verisoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.
14. K. Havelund. Java Pathfinder, a translator from Java to Promela. In *Proc. of SPIN'99*, 1999.
15. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
16. R. Iosif. Symmetry reduction criteria for software model checking. In *Proc. 9th SPIN Workshop on Software Model Checking*, volume 2318 of *LNCS*, pages 22–41, July 2002.
17. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proc. of the 8th international SPIN workshop on Model checking of software*, pages 80–102, Toronto, Canada, 2001.
18. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
19. P. C. Mehlitz, W. Visser, and J. Penix. The JPF runtime verification system. Online manual. `http://javapathfinder.sourceforge.net/JPF.pdf`.
20. M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *SPIN*, pages 28–42, 2005.
21. M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proc. of The First Symposium on Networked Systems Design and Implementation (NSDI)*, pages 155–168, 2004.
22. M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, pages 75–88, December 2002.

23. C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 504–527, Glasgow, Scotland, July 2005.
24. D. Y. Park, U. Stern, J. U. Skakkebæk, and D. L. Dill. Java model checking. In *Proc. of IEEE ASE'00*, 2000.
25. C. E. Perkins, E. M. Belding-Royer, and S. Das. Ad hoc on demand distance vector (aodv) routing, January 2002. IETF Draft.
26. C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100. IEEE Computer Society Press, 1999.
27. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, 2003.
28. A. Sobeih, M. Viswanathan, and J. 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.
29. A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *LNCS*, pages 235–250, 2005.
30. D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.
31. 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.
32. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proc. of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 273–282, New York, NY, 2005. ACM Press.
33. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, 2000.
34. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.
35. W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red-black trees using abstraction. In *Proc. of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 414–417, 2005.
36. W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *Proc. 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2006.
37. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th ASE*, pages 196–205, Sept. 2004.
38. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th TACAS*, pages 365–381, Apr. 2005.
39. J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI*, pages 273–288, 2004.