State Extensions for Java PathFinder

Tihomir Gvero
Milos Gligoric
University of Belgrade
Belgrade, Serbia

Steven Lauterburg
Marcelo d'Amorim
Darko Marinov
University of Illinois
Urbana, IL 61801, USA

Sarfraz Khurshid
University of Texas
Austin, TX 78712, USA

ABSTRACT

Java PathFinder (JPF) is an explicit-state model checker for Java programs. JPF implements a backtrackable Java Virtual Machine (JVM) that provides non-deterministic choices and control over thread scheduling. JPF is itself implemented in Java and runs on top of a host JVM. JPF represents the JVM state of the program being checked and performs three main operations on this state representation: bytecode execution, state backtracking, and state comparison. This paper summarizes four extensions that we have developed to the JPF state representation and operations. One extension provides a new functionality to JPF, and three extensions improve performance of JPF in various scenarios. Some of our code has already been included in publicly available JPF.

Categories and Subject Descriptors: D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.4 [*Software Engineering*]: Program Verification

General Terms: Performance, Verification

Keywords: Java PathFinder, JPF, delta execution, mixed execution

1. INTRODUCTION

Model checking is an important approach for finding bugs. While the original success of model checking was in checking properties of hardware or abstract models of software, recent years have seen a variety of new model checking tools that can directly check programs written in commonly used languages. Some such tools are Bandera, BogorVM, CHESS, CMC, JCAT, JNuke, JPF, SpecExplorer, and Zing.

JPF (from Java PathFinder) [6] is an explicit-state model checker for Java programs. It is the first open-source tool released by NASA, publicly available for download from SourceForge [1]. The release has spawned a growing community of JPF users and contributors, and JPF is becoming an increasingly popular model checker for both research and teaching.

JPF takes as input a Java program (and an optional bound on the length of program execution) and explores all executions (up to the given bound) that the program can have due to different non-deterministic choices, e.g., in thread interleavings. JPF generates as output executions that violate given properties, test inputs for the given program, or statistics about state-space exploration [6].

To support non-deterministic choices, JPF implements a backtrackable Java Virtual Machine (JVM). JPF is itself implemented in Java and runs on top of a host JVM. JPF encodes the JVM state of the program being checked with a special state representation, different from the native representation used by the host JVM. JPF provides three key operations on this state representation: (1) execution that manipulates the state to execute the program bytecodes, (2) backtracking that stores/restores state to backtrack the execution during the state-space exploration, and (3) comparison that detects cycles in the state space.

To increase the applicability and performance of JPF, we have developed several extensions for the JPF state representation and operations [3–5]. This paper summarizes four extensions, which we call *Untracked State*, *Undo Backtracking*, *Delta Execution* [4], and *Mixed Execution* [5]. They affect the state operations in JPF: Untracked State and Undo Backtracking affect backtracking, Mixed Execution affects execution, and Delta Execution affects all three key operations. While Untracked State provides a new functionality to JPF (marking parts of state not to be backtracked), the other three extensions improve performance of JPF in various scenarios; our experiments on a number of data structures and a network protocol show a speedup in overall exploration time from a few percent up to over two orders of magnitude, specifically speedups of 1.42x–6.62x for Undo Backtracking, 0.88x–126.80x (a ratio less than 1x is a slowdown) for Delta Execution [3], and 1.01x–1.73x for Mixed Execution [3].

The rest of this paper first provides more detailed background on JPF and then describes our implementations of the four extensions. While Untracked State and Undo Backtracking were explored previously, most notably for the SPIN model checker, our implementations address dynamic aspects of Java in the context of JPF. We originally proposed Delta Execution [4] and Mixed Execution [5], and detailed descriptions are available in d'Amorim's PhD thesis [3]. We point out that our code for Untracked State has been recently included in the JPF codebase [1], and our code for Undo Backtracking is being considered for inclusion by the JPF core team. In the future, we would like to polish and make publicly available our implementations of Delta Execution and Mixed Execution. We would also like to explore the interplay among the last three extensions since they all target performance of JPF and, in principle, can be used together.

2. BACKGROUND ON JPF

We next introduce the aspects of JPF relevant for our presentation. We use a simple example commonly used to illustrate JPF techniques [2–5]. Figure 1 shows a part of a simplified class `TreeMap` that implements a map interface using red-black trees. Each `TreeMap` object represents a map, and each `Entry` object represents a map element that stores a key-value pair. The map provides standard methods for inserting pairs into the map, removing them, and querying the value for a given key. We can use JPF to explore various tree states that sequences of these methods can reach.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

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

// input bounds sequence length and range of input keys
static void main(int N) {
    // an empty tree, the root object for exploration
    TreeMap t = new TreeMap();
    for (int i = 0; i < N; i++) {
        int methodNum = Verify.getInt(0, 2);
        switch (methodNum) {
        case 0: t.put(Verify.getInt(1, N), 0); break;
        case 1: t.remove(Verify.getInt(1, N)); break;
        case 2: t.get(Verify.getInt(1, N)); break;
        }
        Verify.ignoreIfPreviouslySeen(t);
        /* incrementCounters(methodNum == 1); */
    }
}
```

Figure 1: Parts of TreeMap code and a driver for exploration of tree states

State backtracking: Figure 1 shows driver code that instructs JPF to explore sequences (up to the given length) of method calls with all parameter values (within the given range). The JPF library method `Verify.getInt(int lo, int hi)` returns a value between `lo` and `hi`, inclusively. It creates a non-deterministic choice point: JPF needs to explore the executions for all values in the range. JPF implements this by storing and restoring the entire JVM state of the program being checked, i.e., not only the state of the tree `t` but also the program counter, local variables on the stack, etc. (An alternative approach would be to re-execute the driver, e.g., VeriSoft uses such approach for C programs.)

State comparison: JPF performs a stateful search and stops an execution path if it encounters a previously seen state. By default, JPF compares entire JVM states (unless an annotation `@FilterField` is used to omit some part of the state). However, our driver uses abstract matching [1], represented with a library method `Verify.ignoreIfPreviouslySeen`, to compare only the state of the tree, namely the state of all objects reachable from the root `t`. JPF compares states based on isomorphism by linearizing object graphs into integers arrays [1, 6].

State representation: To efficiently store/restore and compare states, JPF uses a special state representation, different from the native state representation. Recall that JPF is a JVM that runs on top of a native JVM. The native JVM represents `TreeMap` objects using native Java objects (which include pointers to other objects) and thus a heap consists of a number of linked objects. In contrast, JPF encodes each object simply as a Java integer array (`int[]`) and based on type information, interprets a field as either a primitive value or a pointer to another object. JPF encodes the entire heap effectively as an array of integer arrays (`int[][]` in Java).

Bytecode execution: JPF is an interpreter for Java bytecode instructions. JPF provides classes that implement the semantics of Java bytecodes by manipulating the special state representation. The goal for this representation is to make the *overall exploration fast* even if it makes *one straight-line execution path slow*. In particular, for bytecodes that modify the state (e.g., write a field of an object or write to a local variable on the stack), JPF clones the integer array that corresponds to the affected entity (either one object or one stack frame).

```
/* @UntrackedField */
static int totalCounter = 0;
/* @UntrackedField */
static int lastRemoveCounter = 0;
static void incrementCounters(boolean isLastRemove) {
    totalCounter++;
    if (isLastRemove) lastRemoveCounter++;
}
```

Figure 2: Incorrect (as is) and correct (uncomment comments) approaches for counters

Model Java Interface (MJI): JPF provides a mechanism for executing parts of application code (such as `TreeMap`) on the host JVM; to quote from the JPF manual [1]: "*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 allows the host JVM to manipulate the JPF state representation, e.g., to read or write field values or to create new objects. MJI is analogous to the Java Native Interface (JNI) that allows parts of JVM execution to be delegated from the JVM into the native code, written in languages such as C or C++. Some of our extensions leverage or replace MJI.

3. UNTRACKED STATE

This extension provides a new functionality to JPF. By default, JPF stores and restores the entire JVM states. Untracked State allows the user to mark that certain parts of the state should not be restored by JPF during backtracking. This feature is useful for collecting some information about *all execution paths* that JPF explores rather than a *single execution path*. A typical example is counting some events or measuring coverage. We next show a detailed example, then present our definition of Untracked State, and finally describe our implementation in JPF.

Example: To illustrate, consider the driver from Figure 1, and suppose that we want to count (1) the total number of method sequences that produce a new state and (2) the number of such sequences that end with a call to the `remove` method. (The latter is interesting because for many data structures, a sequence that ends with `remove` always results in an old state, but not so for `TreeMap`.) We need to uncomment and implement `incrementCounters`. Figure 2 shows an incorrect attempt to add counters; it does not work since JPF restores the counters when it backtracks the state and thus they would count the number of events on *one path* not on *all paths*.

We introduce a new Java annotation, `@UntrackedField`, that can be used to mark some fields as *untracked*, i.e., not to be restored during backtracking. Uncommenting the two comments in Figure 2 results in a solution that does get the intended values for the counters. Before we added `@UntrackedField` to JPF, the only way to maintain state not backtracked by JPF had been to use MJI (see Section 2). However, MJI requires (i) marking the `incrementCounters` method as `native` and (ii) providing a separate class that implements this method, which results in a longer and non-elegant solution. Due to space limits, we cannot show this solution, but we quote a commit message from Peter Mehlitz, a member of the JPF core team: "*[`@UntrackedField` is] much better than the crude MJI based counters we used for testing so far, or any ad-hoc application solutions.*" We point out that `@UntrackedField` is not just for counters; to quote Peter Mehlitz again: "*@UntrackedField is quite useful for [...] Coverage analysis. For instance, I had this case [...] where I need a collection of objects rather than a counter.*"

Definition: We discuss issues that arise from aliasing when reference fields are marked as untracked. Our implementation allows both static and non-static fields, as well as primitive and reference fields, to be marked as untracked. An object is untracked if all its fields are untracked. If an untracked reference points to an object, that object and all objects reachable from it are untracked. The rationale for this can be seen from the following example: consider that we mark a field of type `String` as untracked; the object that the field points to contains an array of characters, and restoring the array state—even if the `String` itself is not restored—would not produce the desired behavior.

Any number of untracked references can point to the same object graph, and both untracked and regular (tracked) references can point to the same object graph, but *untracked references take precedence over tracked references*: if an object is reachable through any untracked reference, it is untracked. If the execution aliases a tracked object through some untracked field, that object and all objects reachable from it will be untracked from there on. If there are both untracked and tracked reference that point to the same object, the execution can remove all tracked references, and the object remains untracked. The execution can remove all untracked references from an object to which no tracked reference points (and thus the object gets garbage collected). However, if the execution removes all untracked references from the object to which there are some tracked references, the object state becomes *undefined*.

Implementation: Our implementation consists of a new package, `gov.nasa.jpf.jvm.untracked`, and several changes to existing classes. We made our changes to minimally affect existing JPF code. In particular, we did not change the way that JPF stores the state: JPF still stores all fields of all objects, even if some are untracked. We only changed the way that JPF restores the state to avoid restoring untracked fields and objects. Our implementation allows one to dynamically change the status of any object to be either tracked or untracked, although we find reasoning about such changes complicated and recommend the users to make each object either tracked or untracked during its entire lifetime. An alternative implementation would be to not even store the fields and objects that are untracked. Our code is integrated in JPF and publicly available [1].

4. UNDO BACKTRACKING

This extension focuses on speeding up backtracking, but it can reduce the execution time as well as the backtracking time. The key idea is to incrementally store and restore states in JPF. By default, JPF at each choice point *stores and restores the entire JVM state* (except for the untracked part enabled by our Untracked State extension): JPF stores the state when it encounters a choice point and restores it whenever it backtracks to that point. For example, in code from Figure 1, JPF stores the state whenever it executes `Verify.getInt` to assign a new value to `methodNum`. In contrast, Undo Backtracking *does not store the state but only keeps track of the state changes* that happen between choice points and restores the state by undoing these state changes.

Example: To illustrate, consider Figure 1 and an execution of the method `remove` that removes value 3 from the balanced tree that contains values 1 to 3. By default, JPF would store the entire tree before the call (effectively serializing/linearizing the object graph consisting of *three nodes* into an array of integers) and restore it after the call to execute another method. With Undo Backtracking, JPF observes the execution of `remove` and builds a list of fields that are written during the execution. In this example, only the `right` field of the node with value 2 is changed from referencing the node with value 3 to being `null`. Thus, Undo Backtracking would need to remember only *one field change*: Undo Backtracking remembers the old value and restores it when the method finishes. (Changes are undone in the reverse order of that in which they were performed, so the list of changes is effectively a stack.) Undo Backtracking reduces the execution time as it does not require JPF to clone the integer array that encodes all fields of an object whose one field is being written to.

Implementation: Undos/redos (sometimes called "(state) deltas", but we do not use that term to avoid confusion with our orthogonal Delta Execution extension) are a known mechanism for restoring state, e.g., for explicit-state exploration or time-traveling debugging. We implemented Undo Backtracking by adding new classes for storing and restoring states. JPF already provides a modular design for these operations, so we did not need to change any existing code for them. However, we had to change the code that executes field writes to add observers for building the undo objects. Our implementation modifies the JPF interpreter; an alternative would be to use instrumentation to observe the field writes [3].

We have submitted our implementation of Undo Backtracking to the JPF developers to obtain comments on our design and code. Currently, our code tracks changes only to heap objects and arrays, and supports only single-threaded code. When our patch is approved for inclusion into JPF, we plan to add support for multithreaded code by keeping track of the changes to object monitors/locks. We also plan to add support for undos on stack frames; currently, our code stores and restores stack frames fully, as done by default in JPF.

Evaluation: We have evaluated Undo Backtracking on ten data structures, similar to that used in Figure 1, but with drivers that generate data structures directly as object graphs rather than through method sequences. Our initial experiments show that Undo Backtracking can speed up JPF from 1.42x to 6.62x. The speedup (or slowdown) from Undo Backtracking depends on the relative ratio of the state size and the number of state changes between consecutive choice points. In our experience, the number of state changes is almost always smaller than the state size, but it would be worthwhile to explore how to dynamically choose between Undo Backtracking or standard JPF store/restore for certain choice points.

5. DELTA EXECUTION

This extension can reduce overall state-exploration time in JPF. Delta Execution exploits the fact that the execution paths for many methods during state-space exploration overlap. The key idea of Delta Execution is to share overlapping parts of multiple executions and to separately execute only those parts that differ [4]. Delta Execution introduces (1) a novel representation for a set of concrete states (called Δ*State*) and (2) a collection of efficient operations for that representation. This extension targets all three key JPF operations (execution, backtracking, and state comparison).

Example: Consider the driver in Figure 1 for the exploration of sequences of method calls over a range of values. In this driver, `t` represents a single `TreeMap` state. Each method/value combination is executed separately against `t` and subsequently against each new `TreeMap` state encountered during the exploration. In contrast, Delta Execution enables the execution of a method/value combination against multiple `TreeMap` states at the same time. By combining multiple individual `TreeMap` objects into a single ΔState, Delta Execution can perform operations on multiple states at once. It *splits* a set of states into separate subsets at branch control points (e.g., if statements), but only when necessary. For example, a split occurs if for one subset of states a branch condition evaluates to true, while it evaluates to false for the other subset. Naturally, Delta Execution performs best when states are combined. Thus, with Delta Execution, the driver would be modified [4] to *merge* all `TreeMap` states that result at the end of each for-loop iteration back into a single ∆State.

Implementation: Our implementation of Delta Execution uses instrumented Java code that executes under a modified version of JPF. (To automate instrumentation of Java code, we developed an Eclipse plugin [3], but it is not relevant for this description.) We extended JPF in a number of ways to support Delta Execution. For example, to support the key operation of splitting mentioned above, we introduced a new *choice generator* [1] internal to JPF. When splitting a ∆State, a branch control point effectively becomes a non-deterministic choice point, since both branches of execution must ultimately be followed for their respective states. The new choice generator facilitates the identification of which subset of states is active during execution. Another example is the use of special ∆Objects as part of our ∆State. ∆Objects represent sets of values across multiple states. For example, the `size` field for `TreeMap` from Figure 1 would be replaced with a special `DeltaInt` object that would represent the value of `size` across all the states in the current ∆State. Special arithmetic and relational operations that perform on these sets of values were implemented as native methods using JPF's MJI interface.

Evaluation: We evaluated Delta Execution using a bounded-exhaustive exploration as shown in Figure 1 and a non-exhaustive exploration based on abstract matching [1]. For bounded-exhaustive explorations of ten simple subject programs and one larger case study, Delta Execution reduced total exploration time from 0.88x to 126.80x (with median 5.60x) [3]. For non-exhaustive explorations of four subject programs, Delta Execution reduced exploration time from 0.92x to 6.28x (with median 4.52x) [3].

6. MIXED EXECUTION

The key idea of Mixed Execution is to execute *some* parts of the program being checked not on JPF but directly on the host JVM [5]. Of the three main JPF operations on state, this extension addresses only execution; JPF still performs backtracking and state comparison as usual. Mixed Execution executes on the host JVM only *deterministic blocks*, i.e., parts of the execution that have no thread interleavings or non-deterministic choices. 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. We also developed and implemented *lazy translation*, an optimization that speeds up Mixed Execution by translating only the parts of the state that an execution dynamically depends on rather than always translating the entire state reachable from a set of roots.

Example: In the `TreeMap` driver from Figure 1, executions of the `put`, `remove`, and `get` methods manipulate the tree (passed as the implicit `this` argument). JPF uses a special JVM state representation to efficiently store/restore and compare states. Without Mixed Execution, JPF executes all three methods on this special representation, which slows down every field read and write. However, note from Figure 1 that JPF needs to store/restore and compare the state of the tree only at the beginning and at the end of these methods, namely each method can execute atomically.

Mixed Execution executes these methods on the host JVM in three steps. First, 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.) Second, Mixed Execution then invokes the method on the translated state in the host JVM. The method execution can then modify this state. Third, 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.

Implementation: We implemented Mixed Execution by modifying the dynamic dispatch in JPF and providing state translations. Our implementation uses MJI in a novel way: while MJI was originally designed for "components that are not property-relevant" (see Section 2), Mixed Execution uses 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. 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. The pseudo-code of the translation algorithms (including lazy translation) and more details about the implementation are available elsewhere [3, 5].

Evaluation: We evaluated Mixed Execution and lazy translation on six subject programs that use JPF to generate tests for data structures, similar to Figure 1. The experimental results show that Mixed Execution can improve the overall time for state exploration up to 1.73x, while improving the time for execution of deterministic blocks up to 3.05x. Additionally, lazy translation can improve the eager Mixed Execution up to 1.35x. We also evaluated Mixed Execution on a case study that uses JPF to find a bug in a fairly complex routing protocol, AODV, and the results show a speedup of up to 1.41x.

Acknowledgments. We thank Peter Mehlitz, Corina Pasareanu, and Willem Visser for help with JPF; Sasa Misailovic and Aleksandar Milicevic for comments on our work; and Ahmed Sobeih for work on mixed execution. This work was partially supported by the NSF awards #CCF-0702680, #CNS-0615372, #CNS-0613665, and #IIS-0438967, and a gift from Microsoft. Milos and Tihomir did a part of this work as undergraduate summer interns at the Information Trust Institute at the University of Illinos.

7. REFERENCES

[1] Java PathFinder web site. `http://javapathfinder.sourceforge.net`.

[2] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS*, 2007.

[3] M. d'Amorim. *Efficient Explicit-State Model Checking of Programs with Dynamically-Allocated Data Structures*. Ph.D., University of Illinois at Urbana-Champaign, Urbana, IL, Oct. 2007.

[4] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA*, pages 50–60, 2007.

[5] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM*, 2006.

[6] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.