StateComparator: Detecting Unbounded Variables Using JPF

Heila Botha
Dept. of Computer Science
University of Stellenbosch,
South Africa
CAIR, Meraka, CSIR
hvdmerwe@cs.sun.ac.za

Brink van der Merwe and Willem Visser
Dept. of Computer Science
University of Stellenbosch,
South Africa
{abvdm, wvisser}@cs.sun.ac.za

Oksana Tkachuk
SGT Inc. / NASA Ames
Research Center
Moffett Field, California
oksana.tkachuk@nasa.gov

ABSTRACT
Model checking software applications can result in exploring large or infinite state spaces. It is thus essential to identify and abstract variables that could potentially take on a large number of values, in order to increase state matching. In this paper we describe a tool we created as an extension to Java PathFinder, called StateComparator, which compares states in the state space to identify variables that should be abstracted.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

Keywords
Model Checking, State Matching, Java PathFinder, Abstraction

1. INTRODUCTION
Model checking is used to explore finite-state systems and find property violations or prove their absence [2]. Software model checkers such as Java PathFinder (JPF) [7] apply model checking to programs written in modern languages with a large number of potentially complex states. Software applications are not inherently finite-state, which leads to non-termination during model checking.

To model check programs, the state space of the application and its environment must be abstracted to a finite-state system. Often in-depth knowledge of a software system is required to identify and abstract the variables causing the state space explosion. Abstracting a System-Under-Test (SUT) to form a finite-state system has been studied before, but software tools developed for this purpose typically require the user to manually identify fields to be abstracted [3, 5]. Bounded model checking can also be used to ensure a finite-state system by limiting the search depth in the state space, but this approach limits the exploration to a small subset of the system's actual behavior.

In this work we focus on identifying variables causing a state space explosion in JPF. Abstracting these variables decreases the size of the state space which allows the model checker to explore more of the system's behavior. In this paper we describe our solution to identifying such variables by detecting the changes in subsequent states of an SUT. We implement the solution as a JPF listener and show four examples where the tool identifies these variables.

2. BACKGROUND
JPF is an open-source analysis engine for Java applications [7]. It is implemented as an explicit state model checker designed to verify Java applications on a custom JVM implemented in Java. JPF explores the state space of the SUT represented by a directed graph consisting of states and transitions. The *state* of an SUT in JPF consists of three main components [4]:

Thread list Stores a list of the threads and their current states. The state includes a thread's unique id and stack trace. The stack trace contains a list of stack frames — one for each method call storing the location in the method (program counter), its local variables and operands.

Static area Stores a list of all loaded classes and the values of their static variables.

Dynamic area Stores a list of all reference (dynamic) objects in the heap. Each entry contains the type of the object, its unique reference id and a map of its fields and/or list items and their values. Reference ids represent memory locations in the heap and are stored as `int` values.

Static variables, local variables and object fields in the state can either have primitive values or store a reference id of another object-entry in the dynamic area.

A *transition* in JPF consist of the list of instructions leading from one state to another. Each *path* in the state-transition graph represents a possible execution of the application. JPF executes the system deterministically until multiple non-deterministic branches of the execution are possible due to thread interleavings (thread choices) or by different environment configurations (data choices). At a choice point, the transition is ended and the current state of the system is stored. Choices are explored non-deterministically in different branches originating from the state which is restored for each branch using backtracking. JPF stops exploring a path when it reaches an end state (termination of the program) or a previously visited state (in which case the following execution has already been explored).

Explicit state model checkers create states on-the-fly. When a new state is stored in JPF, the current list of threads, classes and the objects in the heap are serialized into a hash value used for quick state matching. To obtain heap symmetry, object entries are stored in the order in which they are referenced and not by their ids. To increase state matching, garbage collection is run before a state is stored to filter out unreferenced objects.

JPF is designed to separate the process of state matching and state storage for backtracking. The serialization of the state into a hash value is only used for state matching. In order to restore the state of the system, JPF stores a memento of the kernel state (thread list, heap, classloaders and listeners) from which the state is restored.

JPF includes an option for state debugging. When enabled, a serialized version of each state is stored in a file. These files can be compared by using a diff tool in order to identify changes in the state.

3. MOTIVATING EXAMPLE

Listing 1 shows a driver for a binary tree implementation we analyze using JPF. The driver adds an integer to the tree and then removes the integer again and asserts that the tree is empty. This code is placed in a while-loop and the transition is ended and a state stored for each iteration, in order to check for state matching. We expect the application to match states after the second iteration of the loop. At this point the binary tree contains no elements — the same state it was in after the first iteration.

```
1  public static void main(String[] args) {
2     BTree<Integer> bt = new BTree<Integer>();
3     while (true) {
4       bt.push(5);
5       bt.remove(5);
6       assert bt.size == 0;
7       Verify.breakTransition();
8     }
9   }
```

Listing 1: Binary Tree example

When we run this example on JPF, however, it does not terminate. This indicates that the state changed for each iteration of the loop. On further investigation and manual state comparison we found that the reason for this is the unbound `modCount` field in the `BTree` class. This field is a modification counter used to catch modifications to the tree while iterating over its nodes using an `Iterator`. The modification counter is increased each time an integer is added or removed from the tree. If we remove `modCount` from the state using JPF's `@FilterField` annotation, the system executes as expected and the state matches after the second iteration of the loop. We should note that `modCount` is used so often in data structures in Java (Lists or Maps) that JPF automatically filters the field from the state for the `java.util` package.

4. DESIGN

Our goal is to identify variables that cause the SUT to have too many states for the model checker to explore. Removing these variables from the state, or abstracting and bounding their values can reduce path lengths and the size of the state space to a more tractable size. Our solution to detect these variables is to track how the state changes along a path in the state-transition graph. These variables' values will continuously change causing states that would normally match to differ.

We implemented our solution as a JPF extension called StateComparator. It consists of a *JPF listener*, detecting which states to compare, and a *state serializer* that caches the state in a `StateInfo` object. `StateInfo` objects are then compared to detect the changes between two states.

4.1 Identifying states

Applications have many states created at different locations in the code and on different paths. These states can only match when the application is in the same location in the code, the threads are in the same state and the state of the heap and loaded classes match. If we compare all states, too many changes can be reported. The user will thus not be able to identify which variables are causing the state explosion. To highlight changing variables, we limit the comparison to states stored at the same location in the code and on the same path in the state-transition graph. Changes detected for these states show the side effects causing states to mismatch at this location.

To mark states for comparison, the user inserts a `markState` statement into the source code:

```
StateComparator.markState(String tagName, int
        startAfter, int stopAfter );
```

The tag name distinguishes between different locations to compare. In this paper we only use a single location (mark-statement) in our examples. To reduce the reported changes and to ensure the analysis terminates the user can also specify the number of marked states after which to start recoding states and the number of marked states after which to stop the analysis.

Each time this statement is executed, the current transition is ended and a new state is stored and marked. The listener serializes the marked state into a `StateInfo` object using the state serializer and caches it until the next marked state is reached. At this point the two states are compared, the changes recorded and the new state cached — replacing the previous state that is no longer needed. To enable the tool to run on a state space with multiple paths, we store and restore (backtrack) the currently cached state.

4.2 Serializing states

Our state serializer is based on JPF's `DebugCFSerializer`. It creates a `StateInfo` object that caches all variable values of a state. We create the `StateInfo` objects during the JPF serialization phase when the entire state is traversed for calculating the hash value of a state for state matching. In this way we save execution time by reusing the traversal of the system state. The `StateInfo` object could also be created when a state is restored by the backtracker. This reduces the number of `StateInfo` objects in memory stored for backtracking, but it increases execution time since the entire state space needs to be traversed again after each marked state is restored (which is currently only done when the state is saved.) Because the user can specify the number of marked states to record and when to start recording, only a few marked states need to be stored to detect changing variables. Therefore the tool does not at this point greatly influence the performance of JPF.

The `StateInfo` object keeps separate maps to cache classes, stack frames and heap objects. The `StateInfo` object stores the unique id and all variable names mapped to their values for classes and heap objects. For stack frames it stores the local variables, the thread id and depth of the stack frame in order to uniquely identify each variable across states. Each variable can either have a primitive value stored in its primitive form (int, boolean, float, char etc.) or store a reference id pointing to another object in the heap. Reference ids are stored as Strings starting with an '@' character to distinguish between them and normal integers or Strings.

4.3 Comparing states

Two states are equal when all their variables (local, static and dynamic) match. Since marked states should match, we are interested in the changes in the state's variables.

Variables with primitive values are compared using a normal Java equals operator. Variables with reference values, however, are more complex. They can be equal even when references differ or unequal when the references are the same. It all depends on the underlying objects to which the references are pointing. To ensure that two objects are truly equal, their fields must be compared recursively otherwise changes may be missed or false changes detected between two objects. This graph of object references can contain cycles when objects reference each other. To avoid cycles during analysis, reference objects are marked as visited when compared and skipped when reached again. The algorithm we use to recursively compare two variables is given in Listing 2. It finds all changes in the object graph. Variable changes are recorded together with the path of objects from the root object to the changed object.

Corresponding local variables can be matched across states using the unique id of its stack frame, the thread id to which the stack frame belongs and the name of the variable. Static variables can also be compared directly using their class's id and the name of the variable, since there can only exist one instance of a class per state.

Matching corresponding dynamic objects in the heap, however, is not directly possible. A simple example of this is the immutable `Integer` wrapper class in Java that cannot be modified. Each time the variable is changed a new `Integer` object with a different reference id is placed on the heap. Although the reference ids differ, it might be the same variable in the program. To match dynamic objects we make use of the same solution implemented by the mark-and-sweep algorithm used by garbage collection to identify all referenced objects in the heap. Instead of comparing dynamic variables directly we recursively compare all *roots* of the objects graphs i.e. the local and static variables. By comparing these variables, we will compare all dynamic variables in the heap. The compare method is called on each of the roots and then recursively on each of their fields/list items.

```
1  boolean compareObjects(obj1, obj2)
2    // to avoid cycles
3    mark obj1
4
5    if obj1 and obj2 is null:
6      return true
7
8    if obj1 or obj2 is null:
9      return false
10
11   if obj1 is a reference id:
12       obj3 = getObject(obj1)
13       obj4 = getObject(obj2)
14
15       if obj3 not marked:
16          return compareObjects(obj3, obj4)
17
18       //already visited
19       return true
20
21   if obj1 != obj2:
22     //primitive variables do not match
23     return false
```

Listing 2: Algorithm used for comparing reference objects

4.4 Interpreting and using output

StateComparator prints the ids of the compared states and for each set of states all variable changes. It also prints the object trace from the root object to the mismatching variable in order to provide context to the variable. This allows the user to identify which variables keep on changing for each state comparison. Each marked state is printed to a file for manual inspection. If a field causing the state space explosion does not influence the property being checked, it can be removed from the state. Otherwise the field must be abstracted to a finite set of values to limit the behavior of the system to reduce the state space.

Variables can be abstracted in several ways using JPF. For example, they can be removed from the state using a `@FilterField` annotation. Filtering values from the state in this way removes the entire object hierarchy of the object from the state used for state matching. The variable still exists within the heap and its state backtracked by the model checker, but it has no effect on state matching. The annotation can also take a condition for when the annotated field should be filtered. The `@FilterField` annotation can be used on static and instance fields but not on local variables. The `@FilterFrame` annotation can filter method stack frames, their program counter and their local variables from the state. Annotations cannot always be added to libraries or application code. To overcome this issue, they can be specified in the `IgnoreConfiguredReflectiveNames` class. JPF also provides `AbstractionAdapters` where fields can be abstracted using a custom method executed when the field is updated. In this case the user implements an adapter for each primitive variable that needs abstraction: local, static or instance variable. When the value of the variable is set, the adapter is fired to ensure that its value stays within the bounds.

5. RESULTS

We evaluate StateComparator on the motivating example above, as well as on three other examples. Applications that have unbounded variables cause non-termination of the model checker. To detect these variables, the search is bounded using a search depth or by specifying the number of marked states after which to terminate the model checker. We also reduce all thread choices in the application to a single random choice to reduce the detected changes using a custom thread scheduling strategy implemented by our tool. Lastly, a `markState` statement (see Section 4.1) is added to the application at a point where the user expects state matching to occur.

A simple example with an infinite number of states is shown in Listing 3. It contains an infinite while-loop that continues to increase the `iTest` local variable for each iteration of the loop. In this example, the model checker does not terminate since `iTest` will never have the same value and so state matching cannot occur.

```
1  public class SimpleExample {
2    public static void main (String[] args) {
3      int iTest = 1000;
4
5      while (true) {
6        iTest++;
7        System.out.println("iTest=" + iTest);
8        StateComparator.markState("TAG1");
9      }
10   }
11 }
```

Listing 3: SimpleExample application

To find the variable causing an infinite state space using our tool, we inserted a `markState` statement on line 8. This statement breaks the transition and marks the new state for comparison. We bound the search space to depth 10 in order to limit the results.

The output of the tool is given in Listing 4. It shows the results of comparing marked states (0, 1), (1, 2), ..., (8, 9). These comparisons show that `iTest` is incremented for each state which causes the states to never match. We limit the `iTest` field to a maximum value of 1002 by extending JPF's `AbstractionAdapter` and rerun the application. Now the loop executed twice before the model checker terminated — the second time the marked state matched the previous marked state and the exploration was stopped.

```
=== COMPARING STATE 1 TO  STATE 0 ===
SimpleExample.main(...)V.iTest:  (1001 ==> 1002)

=== COMPARING STATE 2 TO  STATE 1 ===
SimpleExample.main(...)V.iTest:  (1002 ==> 1003)

=== COMPARING STATE 9 TO  STATE 8 ===
SimpleExample.main(...)V.iTest:  (1009 ==> 1010)
```

Listing 4: Tool output for SimpleExample

Our second application is the "oldclassic" example in jpf-core, inspired by a concurrency defect found on a space craft controller [7]. The application consists of two threads (`FirstTask` and `SecondTask`) that interact by exchanging events. An extract from the code is shown in Listing 5. The `SecondTask` starts by signaling `event1` that wakes up the `FirstTask` waiting on `event1`. When `FirstTask` is notified of `event1` it signals `event2` notifying the `SecondTask` of `event2`. A task can be signaled of a new event before performing a costly wait operation. To optimize the application, each thread caches a copy of the event counter associated with the event on which it waits. If the event counter is increased before it starts to wait, it skips the waiting operation and instead processes the event.

```
1  class Event {
2    int count = 0;
3    public synchronized void signal_event () {
4      count++; notifyAll();
5    }
6    public synchronized void wait_for_event () {
7      try { wait(); } catch (InterruptedException e) {}
8  }}
9
10 class FirstTask extends Thread {
11   Event event1; Event event2;
12   int   count = 0;
13   ...
14   @Override
15   public void run () {
16     count = event1.count;      // caches counter
17     while (true) {
18       StateComparator.markState("TAG1");
19
20       // waits if no event1 has been received
21       if (count == event1.count) {
22         event1.wait_for_event();
23       }
24       count = event1.count;
25       event2.signal_event(); // updates event2.count
26 }}}
27
28 class SecondTask extends Thread {... }
```

Listing 5: "oldclassic" example of jpf-core

The model checker never terminates on this example so we added a `markState` statement in the while-loop of the `FirstTask`. We expect states to match after a few iterations of this loop since no new behavior will be explored. The StateComparator detects four variables changing for each state comparison: the count variables. The changes recorded for the last two states are shown in Listing 6.

These variables can be bounded to a maximum value, in the same way as the previous example, in order to enable state matching.

```
FirstTask.count:  (11 ==> 12)
Object trace:
@163 object FirstTask, mFields={count=12, event1=@15e,
     event2=@15f,...}
@1   frame FirstTask.run(...) locals={this=@163}

Event.count:  (11 ==> 12)
Object trace:
@15f object Event mFields={count=12}
@163 object FirstTask mFields={count=12, event1=@15e, event2
     =@15f,...}
@1   frame FirstTask.run(...) locals={this=@163}

SecondTask.count:  (11 ==> 12)
Object trace:
@175 object SecondTask mFields={count=12, event1=@15e,
     event2=@15f,...}
@1   frame SecondTask.run(...) locals={this=@175}

Event.count:  (12 ==> 13)
Object trace:
@15e object Event mFields={count=13}
@1   frame oldclassic.main(...) locals={args=@bb, new_event1
     =@15e, new_event2=@15f, task1=@163, task2=@175}
```

Listing 6: Changes detected for the last state comparison in oldclassic example

To detect the unbounded `modCount` variable in the binary tree example in Section 3, we replace line 8 with a `markState` statement. To bound the application we set a search depth of 10 and run the application through JPF enabling StateComparator. The changes detected by comparing the first two states are given in Listing 7. Here we can see that the `modCount` field of the binary tree object, defined as a local variable in the main method, is incremented by two for each loop iteration (once for adding an integer value and once for removing it). To reduce the state space we use JPF's `@FilterField` annotation to remove this field from state matching. When run again, the application state matches after the second iteration of the loop.

```
==== COMPARING STATE 1 TO  STATE 0 ====
BTree.modCount:  (2 ==> 4)
Object trace:
@15b object BTree mFields={elements=@15f, modCount=4, size
     =0}
@1    frame BTree.main(...)V: locals:{args=@bb, bt=@15b}

=== COMPARING STATE 2 TO  STATE 1 ===
BTree.modCount:  (4 ==> 6)
Object trace:
@15b object BTree mFields={elements=@15f, modCount=6, size
     =0}
@1    frame BTree.main(...)V: locals={args=@bb, bt=@15b}
...
```

Listing 7: Changes detected for BinaryTree example

The last example is a RSSReader Android application displaying the RSS feed entries to the user. The environment of the application is modeled in JPF-Android [6], an extension to JPF for Android applications. JPF-Android always returns the same set of feed items when the update button is pressed. These items are shown in a list displaying the name and the elapsed time since an item was posted. We expect the application to match after a few presses of the update button, but instead the model checker does not terminate. To identify the problem we analyze a single path in the application and compare the application state after each update button press using StateComparator. The changes detec-

ted for each state comparison are shown in Listing 8. We see that the char[] representing the text in the mText field of the TextField object is changing continuously. On further inspection we found that this TextField stores the elapsed time since the feed item was posted and thus will never be the same since the current time changes — even in JPF. We excluded this field from the state using a @FilterField annotation. Afterwards the application state matched after two presses of the update button.

```
mList:  [-,1,4,7,0,0,6,2,1,9,5,5,7,3,]
==> [-,1,4,7,0,0,6,2,1,9,6,9,1,8,]

Object trace:
@5d0a object char[] mList=[-,1,4,7,0,0,6,2,1,9,6,9,1,8,]
@5d09 object java.lang.String, mFields={value=@5d0a,...}
@5cb8 object android.widget.TextView, mFields={mText=@5d09
    ,...}
...
```

Listing 8: Results for RSSReader example

These examples show how our tool can detect state changes and report them to the user. But, due to space limitations, we cannot show the entire implementation for the last example highlighting the usefulness of the tool in identifying unbounded variables in a large system that contains thousands of variables.

6. RELATED WORK

Previous work has been done on abstracting the environment of Java applications for analysis purposes [5, 3]. Our work focuses only on detecting and bounding fields causing an infinite or too large state space. These fields are hard-to-find in large systems with thousands of variables.

VarTracker, a listener in jpf-core's [1] gov.nasa.jpf.listener package, counts the number of states for which fields, local variables and static variables change. Variables that often change can indicate that they are unbounded or have too many possible values. Although this tool can identify that iTest, the local variable in the SimpleExample (Listing 9), is changing it cannot distinguish when a variable changes back to previous value — not hindering state matching. If we assign iTest = (iTest + 1) mod 3, for example, the same changes will be detected for each iteration if state matching does not occur.

```
change     variable
---------------------------------------
1000   SimpleExample.iTest
1      sun.misc.Unsafe.theUnsafe
1      SimpleExample.main([Ljava/lang/String;)V.se
...
```

Listing 9: Results of VarTracker for SimpleExample

Changes to fields are recored for all instances of a class. If the application contains many of the same objects, the changes will accumulate quite fast for these fields, although they may not be unbounded variables. Lastly, the larger the program is, the more variables change continuously which makes it hard to distinguish which variables should be bounded — especially when variables depend on each other. In the case of the binary tree, for example, the tool detects that modCount is modified, but it also detects that size and many other variables also change (see Listing 10).

```
change    variable
---------------------------------------
10    BTree.remove(Ljava/lang/Object;)V.ii
10    BTree.ensureCapacity(I)V.ii
10    BTree.modCount
10    BTree.size
10    BTree.remove(Ljava/lang/Object;)V.found
 1    sun.misc.Unsafe.theUnsafe
 ...
```

Listing 10: Results of VarTracker for BinaryTree

7. CONCLUSION

Model checking suffers from the state space explosion problem. This problem can be addressed by reducing the number of states and the size of states to increase state matching. This can be accomplished by abstracting the SUT and its environment by bounding variables with too many possible values or by removing variables from the state that have no influence on the property being checked.

In this work we show how variables hindering state matching can be detected by comparing subsequent states in a path, where the application is expected to match. These changes highlight the variables stopping state matching and show how these variables change from state to state. These variables can be abstracted using JPF's abstraction mechanisms.

False positives can be reported by StateComparator when the application contains variables stopping state matching as well as variables alternating between a set of values that would normally allow state matching. Additionally, it cannot detect indirect dependencies between variables. Abstracting one of the detected variable in this case may lead to multiple variable abstractions.

8. REFERENCES

[1] Java Pathfinder. http://babelfish.arc.nasa.gov/hg/jpf/jpf-core.

[2] E. M. Clarke. *The Birth of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[3] J. Hatcliff, M. B. Dwyer, C. S. Păsăreanu, and Robby. Foundations of the bandera abstraction tools. In *The Essence of Computation: Complexity, Analysis, Transformation*, pages 172–203. Springer Berlin Heidelberg, 2002.

[4] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[5] O. Tkachuk. OCSEGen: Open components and systems environment generator. In *Proceedings of the 2nd International Workshop on State Of the Art in Java Program analysis (SOAP)*, number 1, pages 2–5, 2013.

[6] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying Android applications using Java PathFinder. *ACM SIGSOFT Softw. Eng. Notes*, 37(6):1, Nov. 2012.

[7] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. In *Automated Software Engineering*, volume 10, pages 203 – 232. IEEE Comput. Soc, 2003.