

Guided Test Visualization: Making Sense of Errors in Concurrent Programs

Saint Wesonga and Eric G Mercer
Brigham Young University
swesonga@byu.net and eric.mercer@byu.edu

Neha Rungta
NASA Ames Research Center
neha.s.rungta@nasa.gov

Abstract—This paper describes a tool to help debug error traces found by the Java Pathfinder model checker in concurrent Java programs. It does this by abstracting out thread interactions and program locations that are not obviously pertinent to the error through control flow or data dependence. The tool then iteratively refines the abstraction by adding thread interactions at critical locations until the error is reachable. The tool visualizes the entire process and enables the user to systematically analyze each abstraction and execution. Such an approach explicitly identifies specific context switch locations and thread interactions needed to debug a concurrent error trace in small to moderate programs that can be managed by the Java Pathfinder Tool.

Index Terms—Model checking, concurrency, debug, test

I. INTRODUCTION

Program performance is no longer resolved with next generation hardware that increases clock frequency as vendors are now building processors with more cores instead of faster system clocks. As a result, running time can only be improved through better algorithms and more concurrency. Unfortunately, when using concurrency, tools for test or debug are not readily available, making it difficult for a programmer to write correct code. The purpose of the tool in this paper is to test concurrent programs with guided model checking and then debug error traces through visualization of the guidance strategy.

Guided JPF (G-JPF) is a search strategy in the Java Pathfinder (JPF) model checker that finds errors in programs through iterative refinement over an abstraction on the original program [1], [2]. Even though model checking and formal verification are usually precise and sound, they do not scale to large programs [1], [3], [4], [5]. G-JPF therefore turns to heuristic guidance strategies to better scale. The G-JPF abstraction is like a statically computed backward slice starting at program locations that represent errors. These locations are either supplied by the developer (e.g. a statement throwing an exception) or obtained from lint checking tools that flag possible issues in a program. The abstraction is used to guide the concrete execution in JPF. If the program locations in the abstraction cannot be reached in the concrete execution, the abstraction is iteratively refined by adding thread interactions at key program locations until an error trace is generated or the resources are exhausted (time and memory). Empirical results show that G-JPF is able to scale to larger programs that contain errors and can generally produce error traces with minimal thread interactions [6].

The tool described in this paper is a post-mortem visualization engine attached to the G-JPF algorithm. The visualization starts with minimal information in the program slice, and in each refinement, it distinguishes the added thread interactions until the error is reached. The visualization includes portions of the program heap, operand stacks, the underlying program slice used by the algorithm, correlation between the source code and the byte code, filters to control included program locations, and the ability to incrementally visualize the initial abstraction to the final error trace.

This tool is applicable in cases where concurrent code is being written, either when developing new concurrent libraries, or when using prepackaged libraries. A developer creating a concurrent container, for example, would test her container using G-JPF with symbolic execution to model input and use the tool described in this paper to debug any generated error trace. The value of the tool is in explaining the error with a minimal set of thread interactions over critical program locations.

We believe the visualization facilitates debugging an error trace involving concurrency by making clear the needed thread interactions on specific program data. The critical interactions typically involve a small subset of the total number of program locations, available threads, and possible execution schedules. In other words, the visualization improves debugging concurrency errors by finding and showing an error trace involving the least number of threads, context switches, and program locations. A complete and in-depth discussion of the G-JPF underlying algorithm can be found in [2]. The visualization presents to the user the input and output of the G-JPF algorithm and also the information generated in the intermediate steps of the G-JPF algorithm.

II. RELATED WORK

A large body of work has been dedicated to the verification of concurrent programs. We present a discussion of the tools directly relevant to Guided JPF. CalFuzzer is a tool for race-directed random testing of concurrent programs that uses the output of imprecise dynamic analysis tools and randomly drives threads to the input locations [7]. However, the comparison in [8] shows that guiding the search through relevant key locations is significantly better than randomly directing the search to the target locations. Dynamic analysis tools such as ConTest use heuristics to randomly add perturbations

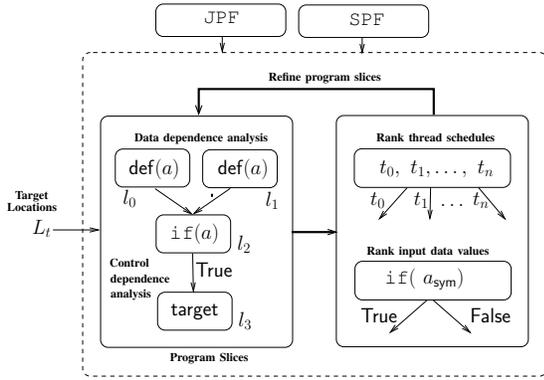


Fig. 1. Overview of Guided JPF

in the thread schedules [9] while Chess systematically explores thread schedules in *C#* programs and supports iterative context bounding [3]. The Guided algorithm, however, has potential to scale to error discovery in even larger programs.

Haveland discusses integrating runtime analysis and model checking in [10]. Although similar in some respects, the Guided algorithm is different because it uses statically computed backward slices to guide its model checking process and therefore does not need an initial complete run of the program.

III. GUIDED TEST OVERVIEW

G-JPF is part of the JPF verification tool-set [11]. JPF is the explicit-state model checker that analyzes Java bytecode with state storage and backtracking capabilities, different search strategies, as well as *listeners* for monitoring and influencing the search. JPF executes Java bytecode based on the standard Java semantics.

In order to visualize the series of events that lead to the error discovery in G-JPF, different pieces of information are recorded during the G-JPF process. This information is then visualized as a post-processing step. Note that the discussion of G-JPF has been simplified here. A more complete and in-depth discussion of the underlying algorithm can be found in [2]. A intuitive overview of G-JPF is shown in Fig. 1.

A. Input

The input to the G-JPF is a set of target locations that represents a possible error in the program being verified. Target locations are either generated from static analysis tools, assertions, or user-specified reachability properties. The lockset analysis in static analysis tools, for example, reports program locations where lock acquisitions by unique threads may lead to a deadlock [12]; the lock acquisition locations are the input target locations to G-JPF. The output of imprecise but scalable static analysis tools such as Jlint, FindBugs, and Chord is ideal to be used as input to G-JPF. In Fig. 1(a), the target location is the program statement marked `ERROR`. The goal of G-JPF is to find a concrete execution that leads to the program location marked `ERROR`.

B. Initial Program Slicing

G-JPF uses program slicing techniques to generate the set of program locations that *may* affect the reachability of the target locations. Intuitively, the goal of these program locations is to serve as guide posts to the possible error. Standard control and data dependence analyses are implemented within G-JPF to generate the backward slices from the target locations. The backward slices contain (1) branch statements whose target outcome affects the reachability of the target location (2) program locations that define global variables that are used in branch statements and affect the reachability of the target locations (3) program locations for synchronization where acquiring or releasing locks affect the reachability of the target locations. The program locations that are part of the backward slice are termed as *key program locations* that determine the reachability of the target locations.

An example of the initial backward slice is shown in Fig. 1(a). The reachability of the location `ERROR` in Fig. 1(a) needs to satisfy the conditions in either abstract trace (i) $x > 10 \implies true \wedge y > 10 \implies false$ or abstract trace (ii) $x > 10 \implies false \wedge z > 10 \implies true$. The G-JPF systematically analyzes each trace in the abstract system in an attempt to find the error along a corresponding concrete execution path. The information from analyzing each trace in the backward slice is recorded in a separate directory to be displayed by the visualization tool. For the example shown in Fig. 1(a) G-JPF will first explore trace (i) and then explore trace (ii) if an error was not found in trace (i). The G-JPF visualization tool does not present the abstract system to the user since it is a graph with many potential paths to the error. Rather the visualization presents the information generated by G-JPF when analyzing a single trace in the initial backward slice that eventually leads to an error trace. The visualization is a postmortem analysis.

C. Search Strategy

G-JPF implements a greedy depth-first search that picks the best immediate successor of the current state and does not consider unexplored successors until it reaches the end of the path. For the example shown in Fig. 1(b), the execution is guided along the trace $1 \rightarrow x > 10 \implies false \rightarrow z > 10 \implies true \rightarrow \text{ERROR}$ in the program slice.

A recording listener within G-JPF records and marks information when the program location of the most recently executed thread in a state matches the program location in the abstract trace of key program locations. The listener stores information about the thread identifier that matched a particular location in the abstract trace. In addition to the matching locations in the abstract trace, the listener also saves other conditional branch statements and data access program statements. In essence the concrete execution path that matches the longest sub-sequence of program locations in the abstract trace is saved by the recording listener. The information stored by the recording listener is used later in the visualization.

The visualized concrete execution correlates to the initial abstract trace, but it includes any executed intermediary byte-

code not included in the initial abstract trace. In other words, the abstract trace includes key byte-codes (source lines) needed to reach the error, and the concrete trace includes every byte-code executed along the path. In addition the visualization also includes values in the heap and the operand stacks. A user is able to traverse the trace, watching the operand stack and heap mutate, up to the point where G-JPF is no longer able to follow the trace due to a unexpected value on a branch condition. The user can switch between a linear trace view such as what is in Fig. 1(b) or the a source code view that gives the completed context of each line of code. Linear trace views and source line views include both the Java source and the associated byte-codes for each source line from the class file. Knobs in the visualization allow a user to jump quickly from locations only in the abstraction, only in the concrete execution, in both the abstraction or concrete execution, or those locations not executed at all. The user is able to see precisely why G-JPF cannot continue execution along the path, and it sets the stage to understand the next step of the algorithm.

D. Guidance Strategies

G-JPF provides a two-level ranking mechanism. In the first level ranking, the states are assigned a rank based on the number of locations in the abstract trace of key program locations that have been encountered along the current execution path. A listener is used to track the locations explored along a path. For the second level of ranking, a configurable heuristic is used to rank states to guide the search toward the next location in the abstract trace.

JPF provides the ability to systematically handle thread scheduling choices. JPF generates a scheduling choice for each enabled thread in the VM state. Distance heuristics are effective in ranking thread non-determinism when guiding the execution toward specific locations in the program. Distance heuristic functions *estimate* the minimal number of transitions required to reach a specified location. The estimate is computed on the control flow representation of the program [13], [8]. For each enabled thread in a VM state the distance estimate from the current program location is computed. The successor state with the lowest estimate is explored before others.

E. Refinement Program Slices

If execution in G-JPF is unable to reach the desired target of a conditional branch statement containing a global variable, then additional inter-thread dependence information is added to the program slice. Suppose, the concrete execution in Fig. 1(b) cannot reach the successor state of where $z > 10$ is *true* because the current value of z is not greater than 10. The refinement process shown in Fig. 1(c) adds another definition of z , by a different concurrently enabled thread, to the trace. The additional program location states that another thread needs to set the value of z before the original thread can continue execution along the original trace. Each refinement incrementally adds thread interactions in an attempt to find the error.

Refinement creates a new abstract trace for G-JPF to follow. G-JPF visualization presents to the user the new refined trace, and it marks by color the new locations added to the trace different from the original abstract trace. If multiple refinements take place, then each refinement adds a new color to the trace. The view of the refined abstract trace quickly focuses the user to context switches on shared variables that affect the reachability of a error location.

As before, G-JPF tries to follow the new refined abstract trace to produce a new concrete execution; G-JPF visualization presents to the user the concrete execution. If it is again unable to reach the target error location, the refinement process is repeated. G-JPF itself systematically analyzes different traces through the initial abstraction of the program until it is able to find an error or exhaust resources (time or memory). The visualization is postmortem, so it only presents to the user the evolution of a trace that leads to an actual error. Such an trace evolution, we believe, aids in debugging as the user can start with an initial minimal set of thread interactions that is easy to conceptualize, and then one by one, add other thread interactions until the error is reachable. With each thread interaction added to the trace, the user is able to incrementally internalize her mental model of the system to debug the root cause of the error.

IV. VISUALIZING GUIDED TEST

We use the program shown in Fig. 2 to demonstrate how the tool visualizes the process of understanding the cause of concurrency errors. This is the *reorder* program from the SIR example set (<http://sir.unl.edu/portal/index.php>). It comprises two Thread subclasses: *CheckThread* and *SetThread* as well as a *SetCheck* class. A main driver class (excluded for brevity) is parameterized to indicate the number of *SetThread* and *CheckThread* instances to create. For simplicity in presentation, we create a single instance of each thread type. The two thread types share a global variable *sc* that is an instance of *SetCheck*.

The visualization tool has two primary presentation modes: A source view that displays a program's Java source code interspersed with its corresponding bytecode as shown in Fig. 3, and a linear trace view that displays a sequence of locations in program order, for example the listings shown in Fig. 4. Fig. 3 is a static view of the program whereas Fig. 4 is a dynamic view of the program's execution. The combination of the source code with the program bytecode in source view can enable the developer to better understand the co-relation between a source code line and its corresponding bytecodes. This co-relation is helpful in the overall understanding of concurrency errors because context switching happens at the byte-code level and not the Java source level, and therefore you need to consider byte-code to fully understand a concurrent trace.

The input to the G-JPF is a set of target locations that represents possible error locations such as assertions or exceptions. In Fig. 2, the target location is the `throw`

```

1 public class SetCheck {
2     private int a=0;
3     private int b=0;
4
5     void set() {a = 1; b = -1;}
6     boolean check() {return ((a==0 && b==0) || (a==1 && b==-1));}
7 }

```

```

1 public class CheckThread extends Thread {
2     SetCheck sc;
3
4     public CheckThread(SetCheck sc) {this.sc=sc;}
5     public void run() {
6         boolean rst = sc.check();
7         if (rst != true)
8             throw new RuntimeException("bug found");
9     }
10 }

```

```

1 public class SetThread extends Thread {
2     SetCheck sc;
3
4     public SetThread(SetCheck sc) {this.sc=sc;}
5     public void run() {sc.set();}
6 }

```

Fig. 2. A listing program that creates 2 threads CheckThread and SetThread which share an object SetCheck where the shared object throws an exception on certain schedules of the two threads.

```

boolean check() {
    return ((a==0 && b==0) || (a==1 && b==-1));
00000 // aload_0 (this)
00001 // getfield a
00004 // ifne 00014 (aload_0)
00007 // aload_0 (this)
00008 // getfield b
00011 // ifeq 00030 (iconst_1)
00014 // aload_0 (this)
00015 // getfield a
00018 // iconst_1
00019 // if_icmpne 00034 (iconst_0)
00022 // aload_0 (this)
00023 // getfield b
00026 // iconst_m1
00027 // if_icmpne 00034 (iconst_0)
00030 // iconst_1
00031 // goto 00035 (ireturn)
00034 // iconst_0
00035 // ireturn
}

```

Fig. 3. An example of how listings are displayed in the tool.

new RuntimeException() statement on line 8 of the CheckThread class.

Note that the SetCheck.check() method returns false (causing the exception to be thrown) if there is an error as determined by the variables a and b. The occurrence of the error in this example is dependent on how the Set and Check threads are scheduled. The goal of G-JPF is to find a concrete execution that leads to that error location, and the goal of the visualization is to aid a developer in understanding the conditions that make the target error location reachable.

The visualization tool does not present the entire abstract system to the user since it is a graph with many potential paths to the error. Rather the visualization presents the information generated by G-JPF when analyzing a single trace in the initial backward slice that eventually leads to an error trace. An example of such an abstract trace is shown in Fig. 4(a), exactly as it is displayed in the visualization tool. Notice, that the target location in the trace is at bytecode position 00019. The difference between Fig. 4(a) and Fig. 4(b) is that Fig. 4(a) is a listing of the locations (in an abstract trace)

```

boolean rst=sc.check();
00000 aload_0 (this)
00004 invokevirtual reorder.SetCheck.check:()Z
return ((a==0 && b==0) || (a==1 && b==-1));
00000 aload_0 (this)
00004 ifne 00014 (aload_0) [Expected Branch: T]
00019 if_icmpne 00034 (iconst_0) [Expected Branch: F]
00027 if_icmpne 00034 (iconst_0) [Expected Branch: T]
00034 iconst_0
boolean rst=sc.check();
00007 istore_1 (rst)
if (rst!= true)
00010 if_icmpgeq 00023 (return) [Expected Branch: F]
throw new RuntimeException("bug found");
00019 invokespecial java.lang.RuntimeException
(a)
00000 invokevirtual reorder.CheckThread.run:()V
boolean rst=sc.check();
00000 aload_0 (this)
00001 getfield sc
00004 invokevirtual reorder.SetCheck.check:()Z
return ((a==0 && b==0) || (a==1 && b==-1));
00000 aload_0 (this)
00001 getfield a
00004 ifne : false 00014 (aload_0) [Expected Branch: T]
00019 if_icmpne 00034 (iconst_0) [Expected Branch: F]
00027 if_icmpne 00034 (iconst_0) [Expected Branch: T]
00034 iconst_0
boolean rst=sc.check();
00007 istore_1 (rst)
if (rst!= true)
00010 if_icmpgeq 00023 (return) [Expected Branch: F]
throw new RuntimeException("bug found");
00019 invokespecial java.lang.RuntimeException
(b)

```

Fig. 4. A simplified abstraction as shown in the tool with an attempt to follow the abstraction in the concrete execution. (a) The initial abstraction of the program in Fig. 2. (b) A concrete execution trying to follow the abstraction but getting stuck at the branch.

that are important in determining the reachability of the target location. As an abstract trace, the locations it includes may be executed by different threads in the concrete execution of the program. It is not necessarily a list of locations to be executed by one thread. The only requirement is that each location listed in Fig. 4 must be executed by some thread. Fig. 4(b) on the other hand shows an initial attempt at executing the program. All executed instructions are grouped together and all the unreachd instructions (from the portion of the abstract trace in Fig. 4(a) that was not reached) are put into the last group.

The visualization annotates every conditional branch with the value of the branch evaluation required to proceed through the entire trace. The first conditional statement in this abstract trace is the ifne¹ bytecode at position 00004. The search for a path to the target location will only succeed if the condition evaluates to true (T) as indicated on that line. When viewing such a trace, it may not be obvious which variables are being examined (because the Java bytecode computing model is stack-based). However, one button click switches to the source view of the check method shown in Fig. 3 and the conditional branches in Fig. 4 can be easily seen in the context of their containing method (since source view is more intuitive for many developers). It is then apparent which variables they operate on as well as which path through the code their branches suggest.

¹Branch execution if the topmost stack operand is not equal to zero.

```

    boolean rst=sc.check();
00000  aload_0  (this)
00004  invokevirtual reorder.SetCheck.check:()Z
    return ((a==0 && b==0) || (a==1 && b==-1));
00000  aload_0  (this)
    a = 1;
00002  putfield  a
    return ((a==0 && b==0) || (a==1 && b==-1));
00004  ifne     00014 (aload_0) [Expected Branch: T]
00019  if_icmpne 00034 (iconst_0) [Expected Branch: F]
00027  if_icmpne 00034 (iconst_0) [Expected Branch: T]
00034  iconst_0
    boolean rst=sc.check();
00007  istore_1 (rst)
    if (rst!= true)
00010  if_icmpeq 00023 (return) [Expected Branch: F]
    throw new RuntimeException("bug found");
00019  invokespecial java.lang.RuntimeException.

```

(a)

```

00000  invokevirtual reorder.CheckThread.run:(V)V
    boolean rst=sc.check();
00000  aload_0  (this)
00001  getfield  sc
00004  invokevirtual reorder.SetCheck.check:()Z
    return ((a==0 && b==0) || (a==1 && b==-1));
00000  aload_0  (this)
00001  getfield  a
    sc.set();
00000  aload_0  (this)
00001  getfield  sc
00004  invokevirtual reorder.SetCheck.set:()V
    a = 1;
00000  aload_0  (this)
00001  iconst_1
00002  putfield  a
    b = -1;
00005  aload_0  (this)
00006  iconst_m1
00007  putfield  b
    return ((a==0 && b==0) || (a==1 && b==-1));
00004  ifne     00014 (aload_0)
00014  aload_0  (this)
00015  getfield  a
00018  iconst_1
00019  if_icmpne 00034 (iconst_0)
00022  aload_0  (this)
00023  getfield  b
00026  iconst_m1
00027  if_icmpne 00034 (iconst_0)
00034  iconst_0
00035  ireturn
    boolean rst=sc.check();
00007  istore_1 (rst)
    if (rst!= true)
00008  iload_1 (rst)
00009  iconst_1
00010  if_icmpeq 00023 (return)
    throw new RuntimeException("bug found");
00013  new java/lang/RuntimeException
00016  dup
00017  ldc "bug found"
00019  invokespecial java.lang.RuntimeException.

```

(b)

Fig. 5. A refined abstract as shown in the tool with a successful attempt to follow the abstraction in the concrete execution. (a) The refined abstraction of the program in Fig. 2. (b) A concrete execution that follows the abstraction and reaches the error location.

G-JPF implements a greedy depth-first search to pick the best immediate successor of the current state. For the example in Fig. 2, G-JPF executes the Reorder program trying to match the trace listed in Fig. 4(a). The resultant concrete execution trace is shown in Fig. 4(b). The initialization and thread creation portion of the concrete trace is omitted for brevity, and the user can control that in the visualization as well.

In any view, instructions that have not been executed in the concrete system are shown with a thread id of -1 because

all executing threads are assigned non-negative thread ids. The other instructions that have been executed in the concrete system are grouped based on the thread that executed them. In Fig. 4(b), for example, all the instructions with locations in the range 00000 - 00004 have been executed by one thread. These thread groupings are shown in different colors to allow the user to easily differentiate groups of instructions belonging to a particular thread as well as context switch locations.

The tool enables the user to select any location in the trace to obtain additional details which are displayed in a “Node Information” tab as shown in Fig. 6(a). For the `ifne` in Fig. 4(b), we can see from the “Node Information” table that it was executed by the thread with ID 2. We can also quickly see the instruction’s class and method (`reorder.SetCheck` and `check`) without switching to source view.

The visualization tool provides filters that allow jumping to certain key locations in the trace such as the first location matching a key location in the abstract trace in Fig. 4(a). In order to determine where execution stopped in a potentially long trace, the user can view the concrete trace listing and set the step through filter to stop only at “Concrete Only” instructions. A “Go to Last Instruction” toolbar button would then highlight the last instruction in the concrete execution.

The `ifne` instruction at 00004 in Fig. 4(b) has a corresponding entry in the abstract system in Fig. 4(a) marking the expected branch outcome as $T(\text{true})$. The branch condition, however, evaluated to false as shown on line 00004 in Fig. 4(b). The previous instruction in Fig. 4(b) is `getfield a` that loads the value of the shared variable `a`. The tool has a “Program Heap” tab shown in Fig. 6(b) that displays the values of the variables in the heap. At the `ifne` instruction we can see in the “Program Heap” tab that the value of `a` is 0 and the branch “`a` is not equal to zero” is false.

The visualization enables us to detect that the value of the variable `a` must not be zero in order for the concrete execution to match the abstract trace. Refinement creates a new abstract trace for G-JPF to follow by adding additional inter-thread dependence information.

The visualization presents to the user the new refined trace. If multiple refinements take place, then each refinement adds a new instruction to the trace. The new instruction is usually shown in a different color to make it stand out to the user. The view of the refined abstract trace quickly focuses the user to *context switches* on shared variables that affect the reachability of an error location. This focus is critical to the user because it becomes immediately obvious which slight scheduling difference can introduce a bug. The refinement in Fig. 5(a) introduces an extra write to the variable `a` into the trace. The different coloring makes it obvious that the `putfield a` instruction has been added to the abstraction.

As before, G-JPF tries to follow the new refined abstract trace to produce a new concrete execution. It systematically analyzes different traces through the initial abstraction of the program until it is able to find an error or exhaust resources (time or memory). As shown in Fig. 5(b), the target location

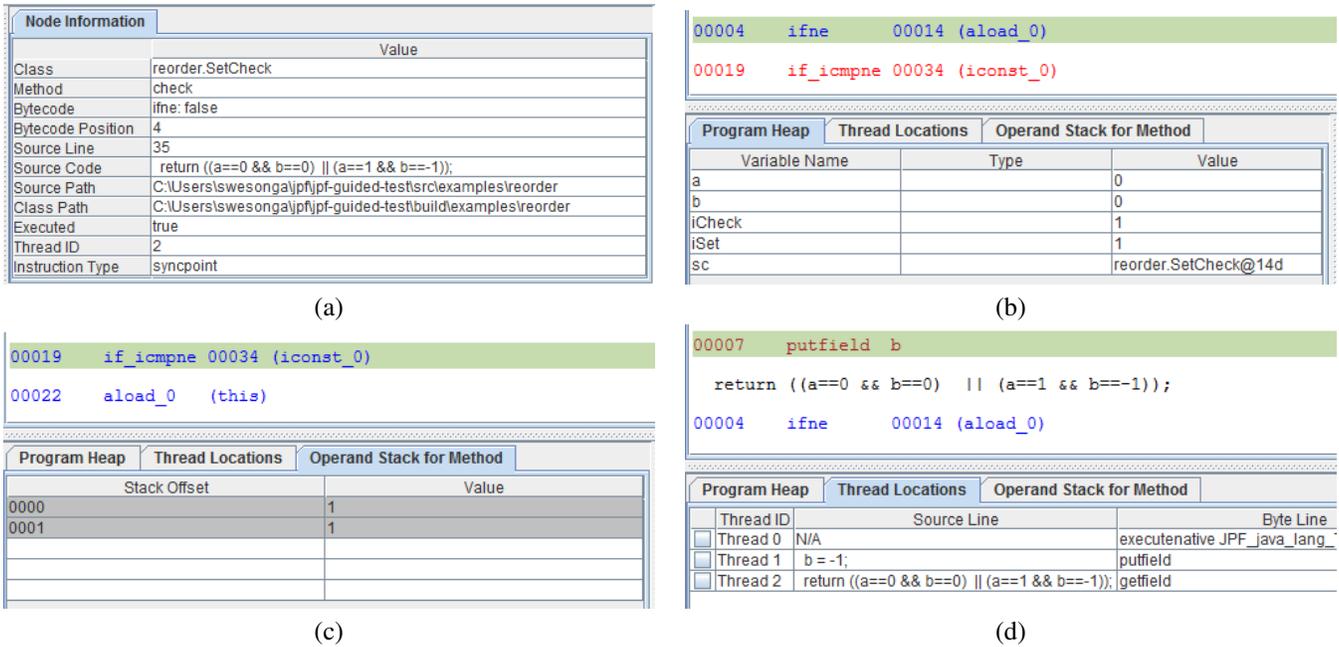


Fig. 6. Different views presented by the tool. (a) Node detail for a byte-code. (b) State variables in the program heap. (c) The operand stack. (d) The current location of each thread.

(the exception) is reached using the refined abstract trace. Consequently, there are no unexplored locations in this final trace - each location has the non-negative thread id of the last thread to execute it.

It is noteworthy that the bug in this program can be fixed by using the `synchronized` modifier on the `set()` and `check()` methods in the `SetCheck` class shown in Fig. 2.

In the new concrete trace in Fig. 5(b), there are instructions (such as the assignments of 1 and -1 to `a` and `b`) that are executed by a different thread before the thread at the `ifne` at location 00004 in Fig. 4(b) resumes. This new thread executed the `putfield a` instruction added in the refinement in Fig. 5(a). As a result of this thread interaction, the `ifne` instruction is able to take its expected branch. The visualization also has thread filters that let the user jump to instructions executed by specific threads.

The tool provides a visual aid to see the values on the operand stack as bytecodes are executed. The `if_icmpne`² bytecode at offset 00027 in Fig. 5(a) expects the true branch to be taken. In the concrete trace in Fig. 5(b), the developer can inspect the actual values that cause the branch condition to match the expected value. With that instruction selected, the “Operand Stack for Method” tab shows what is on the stack when it is executed as shown in Fig. 6(c). The operands used by the selected instruction are highlighted in order to clarify to the developer the effects of the instruction on the stack.

Since the most important events in the search for an error are context switches, it is useful to know where each thread is at any point in the trace. A “Thread Locations” tab shows the current position of each thread as shown in Fig. 6(d). It shows

the concrete trace in Fig. 5(b), when thread 1 is executing the `putfield` instruction. It is then apparent that thread 2 is parked at a `getfield` instruction since `putfield` is updating a variable to aid in the reachability of the target location.

The tool also features playback controls that can be set to automatically step through a trace at a configurable rate making it easy to see the progression and evolution of the trace through the different source lines.

V. IMPLEMENTATION DETAILS

As mentioned earlier, the G-JPF algorithm is discussed extensively in [2]. In order to enable capturing G-JPF’s traces for visualization, two additional run configuration options are supported. A `track_output` flag turns on capturing of trace information and a `traceset_output_loc` configuration option determines where trace files will be saved. Like all other configuration options, these can be used in a `.jpf` configuration file, or in an Eclipse run configuration.

A. G-JPF Search Listener

A listener is implemented that performs various tasks of the G-JPF algorithm. The listener is a subclass of the `JPF PropertyListenerAdapter` class. The listener can monitor and track various JPF-search and JPF-VM related events. For example, it can monitor state advanced, state backtracked, instruction executed and many other events. The primary objective of G-JPF is to find a concrete execution that matches a given abstract trace. To facilitate accomplishing this goal, the listener tracks all program instructions executed along a path generated by a greedy depth-first search. The `searchStarted` method in G-JPF’s listener, part of

²branch if topmost two stack operands are not equal

```

2   public void searchStarted(Search search) {
3       ...
4       abstractTraceLocs = ((GuidedDFSearch)search).
5           getMetaHeuristicInstance().
6           getTraceSet().getAllLocations();
7
8       visitedAbstractTraceLocs = new ArrayList<KeyLocation>();
9
10      oldState = Integer.MAX_VALUE;
11      TrackOutputInstructionVisitor.initializeOnSearchStarted();
12  }

```

Fig. 7. Retrieving the initial set of abstract trace locations.

which is shown in Fig. 7 handles this. It gets the set of abstract trace locations using the main G-JPF algorithm class (GuidedDFSearch) on line 4 then creates a new list to track these locations on line 8.

The `instructionExecuted` notification is generated after every bytecode instruction of the system under test is executed by the JVM. The G-JPF listener monitors this event as shown in the `instructionExecuted` method outlined in Fig. 8. In order to determine which thread is currently executing, it starts with a call to the `JVM.getCurrentThread()` method. It then tracks the class, method, and byte-code position of the most recently executed (and therefore of every) instruction executed in a non-matched state. A non-matched state is one in which back-tracking is not necessary and this is determined by the call on line 4 to the `SystemState().isIgnored()` method. The most recently executed instruction executed is determined with a call to the JVM's `getLastInstruction()` method. Each `Instruction` object has a reference to its containing method's `MethodInfo` object (each method in JPF is represented by a `MethodInfo` object).

To determine the class and method names as well as the argument type names, the corresponding methods are invoked on the `MethodInfo` object on lines 8-10. The information about the different instructions executed is stored in a static data structure (`TrackLocations`) as shown on line 18. If the last instruction executed is a conditional branch statement, it saves the branch condition value of the instruction—`true` or `false`. It also saves a reference to the instruction in the `ProgramLocation` object that was added to the tracker. This is done on lines 21-26. Conditional branch instructions and their actual outcomes (`true` or `false`) are integral to the operation of the algorithm. By saving a reference to the actual instruction we can query and analyze the runtime state to determine whether the branch outcome at the conditional statement matches the expected branch outcome specified in the abstract trace. In the case that the branch outcome of the conditional branch in the concrete trace does not match the one in the abstract trace, the tracked instruction provides information about the point in the executed program where the search cannot proceed.

The final task in the `instructionExecuted` method of the listener is to invoke a specialized visitor class to determine the appropriate information to record for specific instructions. The visitor used for this is the

```

2   public void instructionExecuted(JVM vm) {
3       ThreadInfo currentThread = vm.getCurrentThread();
4
5       if (!vm.getSystemState().isIgnored()) {
6           Instruction lastInstruction = vm.getLastInstruction();
7           MethodInfo mi = lastInstruction.getMethodInfo();
8
9           String className = mi.getClassName();
10          String[] argsTypes = mi.getArgumentTypeNames();
11          String method = mi.getName();
12
13          for(int argIndex = 0; argIndex < argsTypes.length; argIndex++) {
14              method += argsTypes[argIndex];
15          }
16
17          Integer position = lastInstruction.getPosition();
18          ProgramLocation pl = new ProgramLocation(className, method, position);
19          TrackLocations.addLocation(pl);
20
21          if (lastInstruction instanceof IfInstruction) {
22              Boolean condVal = ((IfInstruction)lastInstruction)
23                  .getConditionValue();
24              TrackLocations.addBranchCondition(pl, condVal);
25
26              // Save the IfInstruction in the program location
27              pl.setInstruction(lastInstruction);
28          }
29
30          ... Invoke the instruction tracker (TrackOutputInstructionVisitor)
31  }

```

Fig. 8. An outline of the listener's `instructionExecuted` method.

```

2   public void executeInstruction(JVM vm) {
3       Instruction nextInstr = vm.getNextInstruction();
4
5       ThreadInfo ct = vm.getCurrentThread();
6       lastInstructionOperandStack = Utils.getOperandStack(ct);
7
8       ExecuteInstructionVisitor.vvm = vm;
9       nextInstr.accept(executeInstructionVisitor);
10  }

```

Fig. 9. Capturing the operand stack before each instruction executes.

`TrackOutputInstructionVisitor` class.

The trace visualization tool is a post-processing technique that runs after the search completes. All the information required for the visualization has to be recorded during the search. One such piece of information is the values on the operand stack before each instruction is executed. The `executeInstruction` method is used to record this information. Like the `instructionExecuted` method, the `executeInstruction` method is also part of the `VMListener` interface implemented by our listener in Fig. 9. The notification for the `executeInstruction` event is triggered *before* JPF executes an instruction. The `executeInstruction` event is triggered before the execution of an instruction while the `instructionExecuted` event is triggered after the execution of an instruction notification. The instruction *to be* executed is referred to as the *next* instruction. The call to `JVM.getNextInstruction()` provides the next instruction to be executed by JPF whereas the actual operand stack for the current thread is returned on line 5 by a utility class. Like the `instructionExecuted` method, the visitor pattern is used to invoke the different methods in the `ExecuteInstructionVisitor` class as shown on lines 7-8 of Fig. 9.

Getting the actual operand stack before an instruction executes is done using the `getOperandStack()` method shown in Fig. 10. The method accepts a `ThreadInfo` object

```

1  public static List<Integer> getOperandStack(ThreadInfo ti) {
2      List<Integer> rv = new ArrayList<Integer>();
3
4      StackFrame topFrame = ti.getTopFrame();
5
6      int topPosition = topFrame.getTopPos();
7      int stackBase = topFrame.getLocalVariableCount();
8      int size = topPosition - stackBase + 1;
9
10     for (int i=0; i < size; i++) {
11         rv.add(ti.peek(i));
12     }
13
14     return rv;
15 }

```

Fig. 10. A listing of the `Utils.getOperandStack` method.

representing the thread whose operand stack is to be retrieved. First, the thread’s top `StackFrame` is retrieved by a call to `ThreadInfo.getTopFrame()`. The JPF `StackFrame` object has an integer array of slots (which is combined storage for locals and operands).

The top index of the operand stack (which points to the last pushed value) and the stack base (the index where the operand stack begins in the array of slots) are then computed on lines 5-6 and used to determine the operand stack size on line 7. This is done so that the appropriate number of calls to the thread object’s `peek` method can be made. Hence we can record the list of the operands on the top stack frame.

This information is then passed to an `ExecuteInstructionVisitor` object as mentioned earlier. The class uses visitor methods for the `GETFIELD`, `PUTFIELD`, `GETSTATIC`, `PUTSTATIC` field instructions and records their operands’ names and types to facilitate the visualization later. Fig. 11 shows how these details are retrieved. Each of these four JPF Instructions is an instance of JPF’s `FieldInstruction` class, which has a `getFieldInfo()` method for retrieving its `FieldInfo` object. The `FieldInfo` class stores the type, name and attribute information of a field. It exposes `getType()` and `getName()` methods to retrieve the field’s type and name respectively. The actual value of the field is determined using the `FieldInfo` object’s `getValueObject()` method. The `ExecuteInstructionVisitor`’s `getValueObject(...)` method called on line 9 retrieves the name and type of the field (lines 15-16). These bits of information are then stored in the listener’s `valueObjectInfo` field (line 9).

Whenever a state advances in the search, the locations in the static data structure `TrackLocations`, along with the thread identifier of the most recently executed thread are written to file and the information in the `TrackLocations` is cleared. A state advanced notification indicates a point in the execution where a new observable state is generated. Several program instructions can be executed between a parent and successor state (state transition). Note that all the instructions executed along a state transition will be for the same thread identifier. Any context switch between threads generates a new state in JPF. For the output tracking/trace generation functionality, the last advanced state and the last

```

1  public void visit(GETSTATIC ins) {
2      ThreadInfo ti = vm.getCurrentThread();
3      if (ti.isFirstStepInsn())
4          return;
5
6      FieldInfo fi = ins.getFieldInfo();
7      StaticElementInfo ei = fi.getClassInfo().getStaticElementInfo();
8      Object valueObj = fi.getValueObject(ei.getFields());
9      valueObjectInfo = getValueObjectInfo(fi, valueObj);
10 }
11
12 protected ValueObjectInfo getValueObjectInfo(FieldInfo fi, Object valueObj) {
13     ValueObjectInfo voi = new ValueObjectInfo();
14     voi.object = valueObj;
15     voi.type = fi.getType();
16     voi.name = fi.getName();
17
18     return voi;
19 }

```

Fig. 11. Capturing a field’s name, type and value.

```

1  public void stateProcessed(Search search) {
2      if (MetaHeuristic.getRefinementValue()) {
3          // Since refinement was needed, also track the IfInstruction at
4          // which execution got stuck
5          Instruction ins = MetaHeuristic.getLastRefinementInstruction();
6          ins.accept(trackOutputInstVisitor);
7      }
8  }

```

Fig. 12. Capturing the instruction at which G-JPF got stuck.

thread id to execute in that state are saved as well to enable retrieval of state information when G-JPF completes. This is achieved by implementing the `SearchListener` interface’s `stateAdvanced` method and calling the JVM’s `getStateId()` and `getCurrentThread()` methods.

At refinement points in the G-JPF, we need to record which conditional branch outcome did not allow the G-JPF algorithm to generate a concrete trace that matches the abstract trace. The `stateProcessed` notification is used to handle this condition. In this case, the last `Instruction` at which a refinement was needed is retrieved and passed to the `TrackOutputInstructionVisitor` as illustrated by the listing in Fig. 12. The `MetaHeuristic` class is responsible for performing refinement, hence the calls to its refinement methods on lines 2 and 5.

When the error is finally found, the visualization also needs to track the final state in which the error occurs. For such scenarios, JPF’s `SearchListener` interface provides the `propertyViolated` method, our implementation of which is listed in Fig. 13. It determines the last state advanced to using the G-JPF listener’s `getLastAdvancedState()` method (line 2) and then saves the corresponding last thread to execute in that state (line 5). Finally, it marks that state as valid for the `TrackOutputInstructionVisitor` to ensure that it is displayed in the output. This is because the tracker tracks instructions in all states since the relevant ones will not be known until the error is discovered.

Since the G-JPF algorithm may get stuck while trying to guide concrete execution along locations in an abstract trace, it is useful to include in the concrete trace file the locations in the abstract trace that were not executed. This greatly simplifies trace analysis in the visualization tool. We extended the `TrackOutputInstructionVisitor` visitor class to

```

2  public void propertyViolated(Search search) {
3      int vmstateid = getLastAdvancedState();
4      TrackOutputInstructionVisitor.updateStateNodeMappings(vmstateid);
5
6      stateToThreadMap.put(vmstateid, getLastAdvancedThreadId());
7
8      // Mark the state as valid to ensure it is displayed in the output
9      TrackOutputInstructionVisitor.addValidState(vmstateid);
10 }

```

Fig. 13. Capturing the instruction at which G-JPF got stuck.

```

1 // ins - The instruction being examined
2 // className, methodName - The instruction's corresponding class and method names
3
4 ArrayList<KeyLocation> keylocations = GuidedListener.getAbstractTraceLocations();
5
6 for (int i=0; i < keylocations.size(); i++) {
7     KeyLocation kl = keylocations.get(i);
8
9     if (kl.getClassName().equals(className) &&
10        kl.getMethodName().equals(methodName) &&
11        kl.getPosition() == ins.getPosition())
12     {
13         // Mark the location as visited in the guided listener
14         GuidedListener.addVisitedAbstractTraceLocation(kl);
15
16         traceType = OutputNodeAttributes.TRACE_TYPE_CONCRETE_AND_ABSTRACT;
17
18         break;
19     }
20 }

```

Fig. 14. Detecting and marking abstract trace locations visited during the search.

mark the abstract trace locations that *did correspond* to program locations in the concrete trace. Fig. 14 shows a code snippet illustrating how to detect and mark the abstract trace locations that have corresponding locations in the concrete trace generated during the search. The loop iterates over all the `KeyLocations` in the abstract trace and checks whether the current instruction matches any of them. If so, it marks that location as visited on line 14.

B. Instruction Listener

The `TrackOutputInstructionVisitor` class is used to track every instruction as it executes. It extends the abstract `InstructionVisitorAdapter` class and records all the necessary information about each instruction. One of the tasks performed by the tracker is to load a bytecode's corresponding source line (if available) to simplify the task of viewing a linear trace file. This eliminates having to load the source files in order to make the connection between bytecode and source code. This is a straightforward process since the `Instruction.getLineNumber()` method returns the line number in the source file, which is then retrieved and stored.

The thread specific program counters are also stored by the `TrackOutputInstructionVisitor` class. This is because the visualization presents information about the location of all threads when an instruction executed for a given thread. Fig. 15 details how the thread locations are determined. The `ThreadLocationInfo` object contains the thread id, bytecode position and source line numbers while also providing a custom `toString()` method to generate the corresponding XML output for the trace file. Therefore, each thread id will have an associated `ThreadLocationInfo`

```

1 HashMap<Integer, Map<Integer, List<OutputNodeAttributes>>> stateToThreadXmlMap;

```

Fig. 16. Declaration of the data structure used to track states, threads, and instructions.

object. The hash map created on line 2 of Fig. 15 serves to store this information.

The JVM provides a `getThreadList()` method (which is called on line 3) to retrieve the list of all threads in the JVM. For each of the `ThreadInfo` objects returned, the `getPC()` method is called on line 11 to get the next `Instruction` to be executed by that thread. For a `EXECUTENATIVE` JPF instruction, a call to `instr.toString()` is made on line 21 to include the actual method name executed by the instruction. Also, JPF uses an artificial `RUNSTART` instruction at the beginning of every thread. It serves as a special marker in JPF of the beginning of a new thread execution but does not cause any changes in program state. Therefore, if any thread's first instruction is `RUNSTART`, line 24 uses the `MethodInfo.getInstructionAt(0)` call to get the first actual instruction to be executed by that thread in order to dump the appropriate mnemonic into the trace file. The instruction mnemonic can then be determined using the straightforward `instr.getMnemonic()` call like on lines 25 and 27.

The visualization uses information recorded from the JPF state. The tracking class maintains a map from state id to thread information. This thread information is in turn a mapping from thread id to a list of instructions executed by that thread. In order to facilitate the generation of trace files, a `OutputNodeAttributes` class is used to store all the relevant information about each instruction. This class provides a `toString()` method that can be called when the trace file is being generated, and it outputs all the thread data such as thread id, instruction executing, and so on, all in the correct format. The structure used by the tracking class is therefore declared as shown in Fig. 16.

C. Trace Output Generation

When any JPF search is complete, the `searchFinished` methods of all registered listeners are called. It is in this method that the G-JPF algorithm's listener generates trace files. Fig. 17 shows how this is implemented. It performs a couple of tasks. First, it determines whether refinement was done by G-JPF (line 6). If so, it then determines which state the refinement was needed in (line 7). It then retrieves the mapping of state to thread-instruction map from the `TrackOutputInstructionVisitor` on line 9. Next, it iterates through the states and for each state, ensures that it is relevant to the trace (line 14), determines the last thread to execute in that state (line 22) and uses that thread id to retrieve the list of instructions tracked for that thread by the `TrackOutputInstructionVisitor` object (line 24).

Since both relevant and irrelevant states (in terms of finding the error) are stored by the instruction tracker, the invalid states

```

2     private HashMap<Integer, ThreadLocationInfo> getThreadLocationInfo(ThreadInfo ti) {
3     HashMap<Integer, ThreadLocationInfo> threadLocInfo = new HashMap<Integer, ThreadLocationInfo>();
4     ThreadList threads = ti.getVM().getThreadList();
5
6     Iterator<ThreadInfo> it = threads.iterator();
7     while (it.hasNext()) {
8     ThreadInfo threadInfo = it.next();
9     ThreadLocationInfo tinfo = new ThreadLocationInfo();
10    tinfo.tid = threadInfo.getIndex();
11
12    Instruction instr = threadInfo.getPC();
13    if (instr == null) {
14    tinfo.bytecodeLineNumber = -1;
15    tinfo.sourcecodeLineNumber = -1;
16    } else {
17    String mnemonic;
18
19    if (instr instanceof EXECUTENATIVE) {
20    // Use the toString method to include the name of the actual
21    // method this instruction executes.
22    mnemonic = instr.toString();
23    } else if (instr instanceof RUNSTART) {
24    // Fetch the first actual instruction
25    instr = instr.getMethodInfo().getInstructionAt(0);
26    mnemonic = instr.getMnemonic();
27    } else {
28    mnemonic = instr.getMnemonic();
29    }
30
31    int sourceLineNum = instr.getLineNumber();
32    tinfo.sourcecodeLineNumber = sourceLineNum;
33    tinfo.bytecodeLineNumber = instr.getPosition();
34    tinfo.bytecodeMnemonic = mnemonic;
35    tinfo.instruction = instr;
36    tinfo.sourceLine = (sourceLineNum == -1) ? "N/A" : getSourceCode(instr);
37    }
38    threadLocInfo.put(threadInfo.getIndex(), tinfo);
39    }
40    return threadLocInfo;
41    }

```

Fig. 15. Generating thread locations.

are ignored except for the case where a refinement was done. Each instruction is stored in an `OutputNodeAttributes` object as previously discussed. The trace output generation process ends by outputting the paths referred to in the traces and writing the trace out to disk.

D. Trace File Format

The output generated by the Guided test algorithm is saved in trace files in a customized XML format. Everything in the XML file is stored in a `graph` tag (the document root). There are three primary tags for holding trace content. The first is the `jpffstate` tag. It represents a JPF state and therefore its attributes store all the state metadata needed in the trace file. Currently, only the state id is stored as an attribute of the `jpffstate` tag. The only tag that a `jpffstate` can contain is the `node` tag, which is used to store all the metadata for the instructions executed in that state. There is one `node` tag for each instruction that was executed.

We record information about the location of all the threads, as well as information about the executed instructions, program heap, and operand stack at the time each instruction was executed. The `node` tag can contain `threads`, `variables`, and `operandstack` tags to store these bits of information respectively. The `Threads` tag encapsulates the state of all the threads when a given instruction was executed and is therefore a container for any number of `thread` tags. The `thread` tags in turn detail the thread id, bytecode program counter, corresponding source line position, the actual bytecode and source code, and the class and method of the instruction at that thread's program counter. The state of the variables (name, type, and value) in the program is stored in the `variable`

tag. The `operandstack` tag has a `values` attribute which is a comma delimited list of the operands required by the instruction under which the tag is placed in the trace. The DTD lists these specifications as well as all required attributes for all the tags. It is therefore a useful debugging aid as well since it is used for validating the trace files to ensure that they are in the exact format described.

E. Shell Integration

The trace visualization tool used to aid in viewing and analyzing the G-JPF algorithm's traces is a Java Swing application. It can therefore be run as a "standalone" application as shown in Fig. 18. However, the guided test model checker can also be launched using the JPF shell. The main idea behind the JPF-shell project is to provide a framework for developing domain specific GUIs for JPF tools. Therefore, the visualization tool is also integrated into the JPF shell as shown in Fig. 19. The key advantage of using it as part of the JPF shell is that the trace output location is already known to the tool and the visualization can therefore be done immediately after verifying a program from the exact same interface. For developers that primarily use the shell, this is a significant usability aid, especially since changing output locations does not affect the process of visualizing the generated traces. When the tool is launched as a stand alone application, the traces have to be manually loaded from wherever they are saved.

It is worth noting that the integration of the visualization tool into the JPF shell is a rather straightforward process. In order to extend the functionality of the JPF shell, there are two primary tasks to be performed. First, a custom class that extends the default JPF shell class (`BasicShell`) is created

```

1  public void searchFinished(Search search) {
2      if (this.trackOutput == false) return;
3
4      startXmlOutput();
5
6      boolean didRefine = MetaHeuristic.getRefinementValue();
7      int refinementState = didRefine ? TrackOutputInstructionVisitor.computeRefinementState() : Integer.MIN_VALUE;
8
9      Iterator<Integer> it = TrackOutputInstructionVisitor.stateToThreadXmlMap.keySet().iterator();
10
11     while (it.hasNext()) {
12         int vmstate = it.next();
13
14         if (!TrackOutputInstructionVisitor.isValidState(vmstate)) {
15             if (!didRefine || vmstate != refinementState)
16                 continue;
17         }
18
19         openStateXml(vmstate);
20
21         Map<Integer, List<OutputNodeAttributes>> map = TrackOutputInstructionVisitor.stateToThreadXmlMap.get(vmstate);
22         Integer validThreadId = stateToThreadMap.get(vmstate);
23
24         List<OutputNodeAttributes> nodes = map.get(validThreadId);
25
26         if (vmstate != refinementState) {
27             if (nodes != null) {
28                 for (int k = 0; k < nodes.size(); k++) {
29                     OutputNodeAttributes attr = nodes.get(k);
30                     xmlOutput += attr.toString();
31                 }
32             }
33         } else {
34             xmlOutput += nodes.get(0).toString();
35         }
36
37         closeStateXml();
38     }
39
40     endXmlOutput();
41     outputXml(getCurrentTraceId() + ".xml");
42     searchesRun++;
43 }

```

Fig. 17. Generating the actual trace files.

in order to override any of the behavior or properties of the BasicShell. Next, a ShellPanel providing the desired user interface and functionality is created.

For our tool, the default BasicShell provides sufficient functionality and does not need to be extended. Since the shell functionality is provided through ShellPanel objects, the main user interface class for the visualization tool extends jpf-shell's ShellPanel class. In this case, the parent container for all the UI is the ShellPanel object itself as opposed to a new separate JFrame needed for standalone applications.

The visualization provides various features to aid in trace analysis, most of which have been discussed in the first part of this paper. The main application GUI is shown in Fig. 18. The view displayed is the source view that displays a program's source code interleaved with its corresponding bytecode. The tool also provides a set of trace playback controls to aid developers in navigating traces when debugging.

VI. FUTURE PLANS

There is a large amount of information generated during the model checking run since the information about the program heap, operand stacks, and the executed bytecode instructions is tracked. For real world programs the large amounts of data can prove to be problematic. To mitigate this, we plan to use the jpf-trace-server³ that stores the executed instructions, program heap and operand stack information in a database. This database can be systematically queried to get information about the concrete execution. The visualization

³<http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-trace-server>

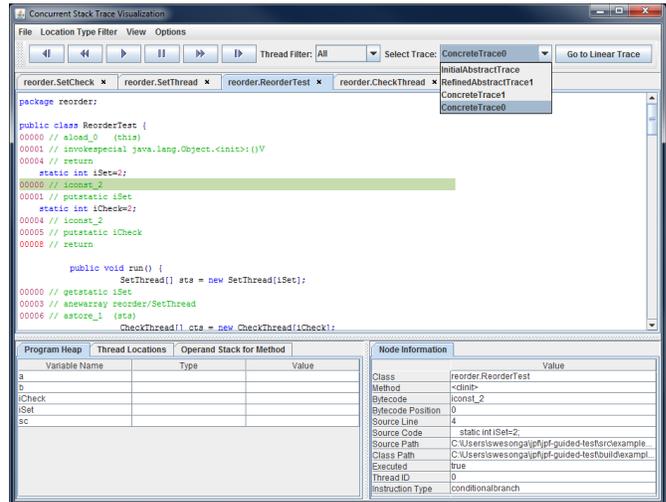


Fig. 18. The visualization tool.

tool will provide the interface to query the jpf-trace-server. Moreover, the trace format used by the tool is not optimal. The generated XML file may contain repetitive information about each program location since these locations are added to the file as they are encountered. The amount of information stored could be reduced during the switch to the jpf-trace-server.

Currently the visualization tool has just been used internally. Another avenue for future work is to perform a user-study in order to evaluate the utility of the visualization in facilitating the debugging of concurrent programs. An interesting

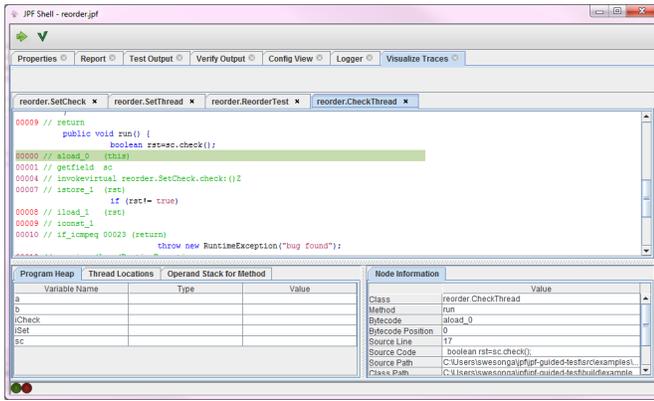


Fig. 19. The visualization as part of the JPF shell.

dimension of the user study would be to evaluate if such a visualization tool can be helpful to students learning about concurrency. A tool like this can potentially be used as teaching aid to help students understand the effects of concurrency and provide intuition on how threads interact.

VII. CONCLUSION

The tool described in this paper helps understand error traces in concurrent programs found through guided model checking. The tool presents to the user the evolution of an error trace containing minimal thread interactions over critical program locations. Such an error trace evolution, aids in debugging as the user can start with an initial set of thread interactions that is easy to conceptualize, and then one by one, add other thread interactions until the error is reachable. With each thread interaction added to the trace, the user is able to incrementally internalize her mental model of the system in debugging the root cause of the error.

The G-JPF tool is available from a mercurial repository at <http://babelfish.arc.nasa.gov/hg/jpf/jpf-guided-test>. The Visualization tool is the `TraceVisualization` program in the `edu.byu.cs.guided.search.visualize` package.

REFERENCES

- [1] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Proc. ASE*, Grenoble, France, September 2000.
- [2] N. Rungta, E. G. Mercer, and W. Visser, "Efficient testing of concurrent programs with abstraction-guided symbolic execution," in *Proc. SPIN Workshop*. Springer-Verlag, 2009.
- [3] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *SIGPLAN Not.*, vol. 42, no. 6, pp. 446–455, 2007.
- [4] P. Godefroid, "Verisoft: A tool for the automatic analysis of concurrent reactive software," in *Computer Aided Verification*, 1997, pp. 476–479. [Online]. Available: citeseer.nj.nec.com/godefroid97verisoft.html
- [5] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan, "Dynamic model checking with property driven pruning to detect data race conditions," in *ATVA*, ser. LNCS. Seoul, Korea: Springer, 2008.
- [6] N. Rungta and E. G. Mercer, "Clash of the titans: tools and techniques for hunting bugs in concurrent programs," in *PADTAD '09*. ACM, 2009, pp. 9:1–9:10.
- [7] K. Sen, "Race directed random testing of concurrent programs," *SIGPLAN Not.*, vol. 43, no. 6, pp. 11–21, 2008.
- [8] N. Rungta and E. G. Mercer, "Guided model checking for programs with polymorphism," in *PEPM*. New York, NY, USA: ACM, 2009, pp. 21–30.

- [9] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 267–279, 2007.
- [10] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *Proceedings of the 7th International SPIN Workshop on Software Model Checking*. London, UK: Springer-Verlag, 2000, pp. 245–264.
- [11] Java PathFinder Tool-set, "<http://babelfish.arc.nasa.gov/trac/jpf/>."
- [12] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proc. SOSP '03*. New York, NY, USA: ACM Press, 2003, pp. 237–252.
- [13] S. Edelkamp and T. Mehler, "Byte code distance heuristics and trail direction for model checking Java programs," in *Proc. MoChArt*, 2003, pp. 69–76.