A Progress Bar for the JPF Search Using Program Executions

Kaiyuan Wang, Hayes Converse, Milos Gligoric, Sasa Misailovic*, and Sarfraz Khurshid
University of Texas at Austin, *University of Illinois at Urbana-Champaign
{kaiyuanw,hayesconverse,gligoric,khurshid}@utexas.edu, misailo@illinois.edu

ABSTRACT
Software model checkers, such as JPF, are routinely used to explore executions of programs that have very large state spaces. Sometimes the exploration can take a significant amount of time before a bug is found or the checking is complete, in which case the user must patiently wait, possibly for quite some time, to learn the result of checking. A progress bar that accurately shows the status of the search provides the user useful feedback about the time expected for the search to complete. This paper introduces JPFBar, a novel technique to estimate the percentage of work done by the JPF search by computing weights for the execution paths it explores and summing up the weights. JPFBar is embodied into a listener that prints a progress bar during JPF execution. An experimental evaluation using a variety of Java subjects shows that JPFBar provides accurate information about the search’s progress and fares well in comparison with a state-based progress estimator that is part of the standard JPF distribution. We implement JPFBar as a JPF listener and it is available at https://github.com/kaiyuanw/JPFBar.

Keywords
Search progress bar; model counting; JPF

1. INTRODUCTION
Software model checking [Godefroid 1997, Visser et al. 2000, Corbett et al. 2000] today is a well-established method for systematically analyzing behavioral correctness of software systems. Modern model checkers, such as Java PathFinder (JPF) [Visser et al. 2000], readily handle complexities of advanced programming language constructs, e.g., JPF handles all of Java bytecode. However, checking complex software systems that have very large state spaces can still take state-of-the-art model checkers that have sophisticated pruning techniques, such as partial order reductions [Clarke et al. 1999], quite some time to report a bug or complete the checking. Due to the underlying complexity of the core model checking problem, during the run of a model checker, it is hard for the user to determine when the run may terminate. Indeed, sometimes the user may be compelled to terminate the analysis out of frustration without gaining much insight into the percentage of work that has been done before the termination.

A progress bar that accurately predicts the state of the run of a model checker can provide a valuable practical tool that not only enhances the overall user experience with their model checker but also provides useful technical feedback. However, designing an accurate progress bar for a software model checker is a challenging problem. A key issue is that for many non-trivial systems the shape of the state space of the system is only known once the analysis is complete, which makes accurately determining how much work is remaining hard.

In this paper we study the problem of defining a progress bar for JPF and report on our work that introduces JPFBar, a novel technique to quantify the work completed by JPF search. Our key insight is that since an explicit-state model checker like JPF actually executes and checks (many) program behaviors, a useful progress bar can be based on the program executions that are explored by the model checker. Specifically, for each complete execution path that JPF explores, JPFBar computes the progress with respect to that path and adds it to the overall progress made so far. Conceptually, JPFBar assigns each path a weight based on the non-deterministic choices in the system. The overall progress is monotonic, i.e., the progress grows monotonically, between 0 and 1 (i.e., 100%). Since JPFBar calculates the progress simply based on the execution paths explored, it naturally supports the different search strategies in JPF, including the various heuristic searches. Moreover, JPFBar handles state space graphs that may not be acyclic, e.g., when two unique paths in the graph lead to the same state, which JPF can detect when state matching is turned on.

We embody JPFBar as a JPF listener on top of the JPF core, which in principle allows it to function with existing JPF extensions. We evaluate JPFBar using a suite of benchmarks that have state spaces with various characteristics, e.g., some have uniform branching and some have highly skewed branching. The results show that JPFBar introduces a promising approach for estimating the model checker’s progress. Moreover, we compare JPFBar with a previous progress reporting technique, namely the StateCountEstimator listener, which is a part of the standard JPF distribution and reports monotonic progress based on the number of states explored. Experimental results show that JPFBar compares well with StateCountEstimator for the standard depth-first search. Also, we find that the StateCountEstimator listener only supports the depth-first search strategy.

This paper makes the following contributions:

• **Path-based progress bar.** It introduces the idea of measuring the progress of an explicit state model checker using the execution paths it explores.
• **Technique.** It introduces JPFBar, a novel progress bar that handles various search strategies and heuristics, as well as state space graphs that may or may not be acyclic. We make JPFBar publicly available at https://github.com/kaiyuanw/JPFBar.
• **Evaluation.** It experimentally evaluates JPFBar to demon-
import gov.nasa.jpf.vm.Verify;
public class TwoChoices {
  public static void main(String[] a) {
    System.out.println(""," + Verify.getInt(1, 3) + ");
  }
}

(a) Example with 2 non-deterministic choices  (b) JPF

Figure 1: Example program and JPF output

Figure 2: State-space graph for the example program.

Figure 3: Progress output – depth-first search

Figure 4: Progress output – breadth-first search

An accurate progress bar can indeed serve as a key tool that enhances the practical usability of model checkers and, more generally, other tools that handle very large exploration spaces, e.g., constraint solvers [Boyapati et al. 2002, Een and Sorensson 2003], decision procedures [de Moura and Bjorner 2008], and program analyzers [King 1976]. We believe JPFBar provides a promising technique for explicit-state model checkers in general and JPF in particular, and in future work can provide a basis for more accurate state coverage estimators that serve as practical metrics for evaluating different search strategies or even checking techniques.

2. EXAMPLE

This section illustrates JPFBar using a simple example that makes two non-deterministic choices (Figure 1a). The TwoChoices example program simply creates two choice generators and prints a pair of integers where the first integer ranges from 1 to 3 and the second integer ranges from 1 to 2. If we run JPF against the example, we will observe 6 pairs of integers (Figure 1b).

Figure 2 shows the state-space graph for the TwoChoices program. Each edge represents a non-deterministic choice and is labeled with (1) the weight that is used to calculate the progress for the paths that contain that edge; and (2) the choice (in square brackets) represented by that edge. Each leaf node is labeled with the weight of the corresponding path, i.e., the incremental progress that JPFBar computes for that path. The graph has 10 states (s0, s1, . . . , s9), of which 6 states (s4, . . . , s9) are end states.

As its default search strategy, JPF uses depth-first search (DFS), which is implemented in class gov.nasa.jpf.search.DFS. Figure 3a and Figure 3b show the JPF output for DFS of the main method using the traditional StateCountEstimator listener and our JPFBar listener, respectively. The StateCountEstimator listener prints the progress information periodically in the format of “State: $X / $Y ($Z%)”, where $X is the number of states explored so far, $Z is an estimated progress percentage, and $Y is computed by $X/$Z; in addition to state information, StateCountEstimator also prints time information, which we omit here. JPFBar prints the progress information periodically in the format of “[Path] $X / $Y ($Z%)”, where $X is the number of paths explored so far, $Z is an estimated progress percentage, and $Y is computed by $X/$Z. For this example, JPFBar’s progress information is more evenly distributed and accurate w.r.t. the program output compared to the StateCountEstimator listener.

In addition to DFS, JPF supports various searches, e.g., breadth-first search (BFS) (in the gov.nasa.jpf.search.heuristic package). Figure 4b shows the JPF output with respect to BFS of the main method using our JPFBar listener. In comparison, the traditional StateCountEstimator listener does not support BFS (or other searches in gov.nasa.jpf.search.heuristic) and outputs just “State: 0 / 0 (100.000%)”.

3. TECHNIQUE

This section presents our JPFBar technique. While our focus here is the Java PathFinder (JPF) model checker, the core JPFBar technique can be applied to other explicit-state model checkers. JPFBar introduces an execution-based technique to report the JPF model checker’s progress. For each complete execution path that JPF explores, i.e., paths that terminate in an end state, JPFBar computes the progress for that path as its weight based on the path’s branching structure with respect to the non-deterministic choices along that path. More precisely, let p be a complete execution path that consists of the following sequence of non-deterministic choices: (c1, . . . , ck), where ci is a choice out of ti total choices. The weight of p, written weight(p) then is

$weight(p) = \prod_{i=1}^{k} t_i \\

JPFBar incrementally computes the overall search progress as each execution path is explored by JPF and reports the progress as a percentage (starting at 0%) at the end of each path.

Since the weight is divided evenly amongst all of a state’s children, at any state s, the weight of its children adds up to the weight of s. We assign the weight of the root to be 1, so the weight of all paths sums to 1.

JPFBar handles state-space graphs that are acyclic as well as graphs that are not acyclic. If the graph is not acyclic and state matching in JPF is turned on, JPF may backtrack along a path before reaching an end state. JPFBar accurately accounts for such backtracking by incrementing the overall progress by the weight of the path that backtracks due to state matching. To illustrate,
all candidate lists and checks the validity of each of them using list nodes; this subject uses non-deterministic choice to initialize invalid singly-linked lists within a given bound on the number of bound.

JPF to count the set of valid and invalid binary tree with a given List JPF distribution.

StateCountEstimator listener implementation.

We embodied JPFBar as the uniformly. We plan to explore this connection in future work.

at each non-deterministic choice point, picks one of the choices ing a random search strategy that begins at the start state and tween 0 and 1 (i.e., 100%). Observe that the weight of each path

Figure 5: State-space graph that is not acyclic. JPF explores three paths: $p_1 : (s_0, s_1, s_3, s_4)$; $p_2 : (s_0, s_1, s_3, s_5)$; and $p_3 : (s_0, s_2, s_3)$. JPFBar calculates the following weights for them: $p_1 : \frac{1}{5}; p_2 : \frac{1}{4};$ and $p_3 : \frac{1}{4}$. The weights add up to 1 and represent 100% progress when JPF search terminates.

Figure 5 shows a hypothetical state space graph that has a cycle, and how JPFBar handles it. Moreover, if a path terminates in an error state, JPFBar updates the progress with the weight of that path to handle such paths.

Overall, the progress reported by JPFBar, i.e., $\sum weight(p)$, where $p$ ranges over all explored execution paths that terminate in an end state, a state match, or an error, takes a numeric value between 0 and 1 (i.e., 100%). Observe that the weight of each path can alternatively be viewed as the probability of executing it using a random search strategy that begins at the start state and at each non-deterministic choice point, picks one of the choices uniformly. We plan to explore this connection in future work.

We embodied JPFBar as the PathCountEstimator listener that we built on top of core JPF. Figure 6 shows the key parts of our listener implementation.

4. EVALUATION

We evaluated JPFBar on 10 subjects and 8 search methods, and compared it to StateCountEstimator, the existing comparable tool. For lack of space, we show our experimental results in scatter plots for 7 subjects and 3 search methods in Figure 7. We observe similar results for other subjects and search methods which are not shown. DiningPhil and Racer are from the standard JPF distribution. List uses JPF to count the set of valid and invalid singly-linked lists within a given bound on the number of list nodes: this subject uses non-deterministic choice to initialize all candidate lists and checks the validity of each of them using an executable check (repOk method). BinaryTree similarly uses JPF to count the set of valid and invalid binary tree with a given bound. LeaningLeft, LeaningRight and LeaningBalanced are programs that we specifically designed to make JPF state exploration graphs that lean towards the left, lean towards the right, and are balanced, respectively. DFSSearch is the default depth-first search strategy. BFSHeuristic is the breadth-first search strategy. RandomHeuristic is the random search strategy. All these search strategies are available in the standard JPF distribution under the package gov.nasa.jpf.search.

For each subject and search method, we gathered data in the form of progress reports from the listeners, and recorded the actual completeness of the search at that time, as defined by the number of actions (StateAdvanced and StateBacktracked) taken so far divided by the final number of actions. In the experiment, we let JPF report all errors instead of stopping the exploration at the first error. We then plotted these points in a scatter plot of reported progress (y-axis) versus actual progress (x-axis) in percent, and computed the Pearson correlation coefficients, or R-values, for these relationships. Red circles rep-
<table>
<thead>
<tr>
<th>Program</th>
<th>DFSearch</th>
<th>BFSHeuristic</th>
<th>RandomHeuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiningPhil</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
<tr>
<td>List</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
<tr>
<td>LeaningBalanced</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
<tr>
<td>LeaningLeft</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
<tr>
<td>LeaningRight</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
<tr>
<td>Racer</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
<td>[Graphs]</td>
</tr>
</tbody>
</table>

Figure 7: JPFBar and StateCountEstimator Results Comparison
previous work is that JPFBar estimates the progress purely based on the information already available during JPF search; for example, JPFBar does not require any additional probes into the search space. In this regard, JPFBar follows the spirit of the StateCountEstimator listener, which is a part of the standard JPF distribution. A key difference is StateCountEstimator's use of state counts and JPFBar's use of exploration path counts to define progress. Moreover, JPFBar works with various JPF search strategies that StateCountEstimator does not handle. We believe probing can play an important role in defining a more accurate progress bar for JPF and we plan to investigate it further.

7. CONCLUSIONS
This paper introduced JPFBar, a novel technique to estimate the percentage of work done by the JPF search by computing weights for the execution paths it explores and summing up the weights. JPFBar is embodied into a listener that prints a progress bar during JPF execution. An experimental evaluation using a variety of Java subjects shows that JPFBar provides accurate information about the search's progress and fares well in comparison with StateCountEstimator, a state-based progress estimator in the standard JPF distribution. We implement JPFBar as a JPF listener and it is available at https://github.com/kaiyuansw/JPFBar.

Acknowledgments
We thank the anonymous reviewers for their comments. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1703637, CCF-1629431, CCF-1704790, CCF-1652517.

8. REFERENCES


