Probabilistic Model Checking with Java PathFinder*

Xin Zhang and Franck van Breugel

DisCoVeri Group, York University, Toronto, Canada

1 Introduction

On the one hand, probabilistic model checkers such as PRISM [1] have been successfully employed to verify models of probabilistic systems. However, they are not suitable for checking properties such as uncaught exceptions of the actual code of the system. On the other hand, model checkers such as Java PathFinder (JPF)[1] [2] have been used with success to verify actual code of systems. However, they do not take into account the probabilities associated with the probabilistic choices of the systems. In this paper, we bridge the gap by extending JPF to a probabilistic model checker.

JPF is an explicit state model checker of Java bytecode. In its basic form, it is a Java virtual machine. In contrast to an ordinary Java virtual machine, JPF systematically explores all potential executions of a system, rather than a single one. Each execution is a sequence of transitions. Each transition consists of a sequence of bytecode instructions. While exploring those executions, JPF tries to find violations of properties like uncaught exceptions and deadlocks. JPF has been designed in such a way that it can be easily extended.

2 Extension of JPF to a Probabilistic Model Checker

If we model check the Java code of a probabilistic system, JPF does not take into account any probabilities associated with the probabilistic choices in the code. We have extended JPF to a probabilistic model checker by associating probabilities to the transitions.

To express probabilistic choices in the Java code, we have introduced the class `Choice` of the package `probabilistic` which contains the static method `make(double[] p)`.[2] Given an array `p` of doubles with $\sum_{i=0}^{\texttt{p.length}-1} \texttt{p}[i] = 1$, the invocation `Choice.make(p)` returns i with probability `p`$[i]$. Hence, the invocation `Choice.make(0.5, 0.5)` returns either 0 or 1, both with probability 0.5.

* This research is supported by NSERC.

[1] `http://babelfish.arc.nasa.gov/trac/jpf/`

[2] Our extension has been developed in a modular way such that we can deal with existing methods such as the `nextBoolean` method of the `Random` class in the same way as we handle our `make` method of the `Choice` class.

An invocation of the `make` method contains the probabilities of the probabilistic choice. JPF needs those probabilities. Hence, JPF has to treat the invocation of this method differently from the invocations of other methods. Therefore, we introduce a so-called model class, named `JPF_probabilistic_Choice` according to JPF's convention for naming model classes, which also contains the method `make`. Whenever JPF encounters an invocation of the `probabilistic.Choice.make` method, it does not model check the bytecode of that method, but it considers the bytecode of the `JPF_probabilistic_Choice.make` method instead. This bytecode provides JPF with the probabilities associated with the probabilistic choice represented by the `make` method. How these probabilities are employed by JPF is discussed next.

3 New Search Strategies

JPF can check the transitions in different orders by using, for example, a depth-first search (DFS) or a breadth-first search (BFS). Since we have extended JPF by associating probabilities to the transitions, we can use these probabilities to determine in which order to explore the transitions. In [3] we introduce several new search strategies which use the probabilities associated with the transitions. These new search strategies have been implemented in JPF. Given the modular way we have extended JPF, new search strategies can easily be added.

To let transitions with the highest probability be searched first, our probability-first search (PFS) strategy sorts the enabled transitions by their probability. Our breadth-first probability-second search (BFPSS) is an enhancement of BFS in which transitions at the same level are sorted by their probability. Our randomized search (RS) randomly selects an enabled transition, where the chance of a transition being selected is proportional to its probability.

4 Measuring the Progress of JPF

In many cases, JPF will either run out of memory or will simply not terminate within any reasonable amount of time when verifying the code of a probabilistic system. In [3], we introduce the notion of a progress measure for model checkers such as JPF.

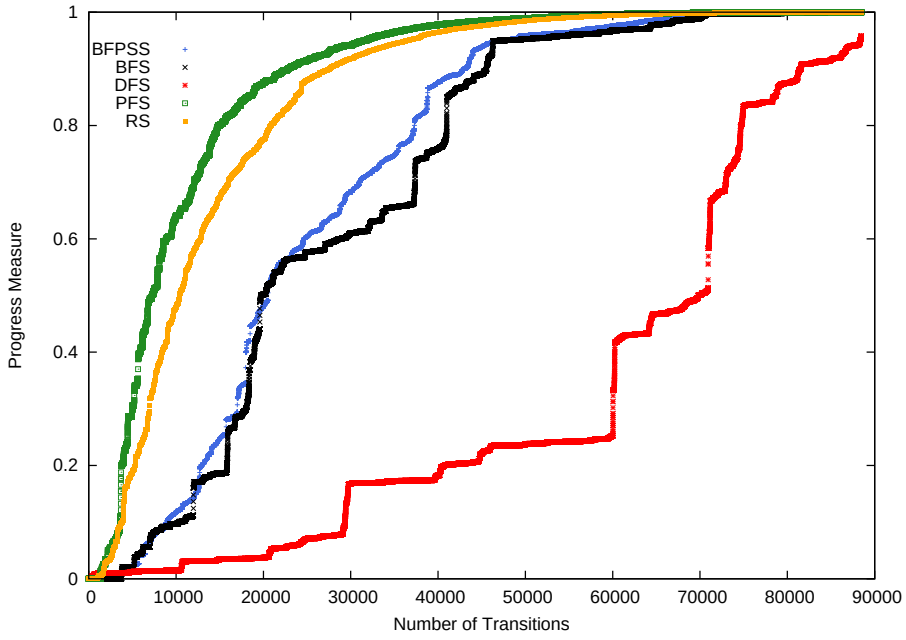
To measure JPF's progress by simply counting the number of executions that have been checked is not very useful for several reasons. First of all, it may be very difficult or even impossible to determine the total number of potential executions. Hence, the number of executions that have been checked by JPF gives us very limited information about the amount of progress that has been made. Secondly, some executions are more likely to happen than others. For example, nonterminating executions often occur with probability zero. Checking such an execution amounts to no progress at all.

Instead of counting the number of executions, we endow the set of potential executions with a σ-algebra and a probability measure. In this way, we obtain a probability space of executions. The measure of the set of executions that

have been checked gives us a number in the interval [0, 1]. This number provides us a quantitative measure of the amount of progress JPF has made. The larger the number, the more progress JPF has made. In [3] we have shown that computing the progress measure can be reduced to computing the measure of the complement of the set of those executions of the probabilistic system that satisfy a particular temporal logic formula. To compute the measure of this set, we can use, for example, the algorithm of Courcoubetis and Yannakakis [4, Lemma 3.1.1.1]. For more details, we refer the reader to [3].

We have implemented the progress measure in JPF. In our implementation we delegate to MRMC [5] to compute the measure of the above mentioned set of executions. After each transition being explored by JPF, the progress is computed. Our extension of JPF can handle millions of transitions.

We have compared the amount of progress the different search strategies make for probabilistic systems. To make such a comparison, we have implemented a number of randomized algorithms in Java. For each Java implementation, we have run JPF with the different search strategies. In the graph below, the amount of progress made by JPF when model checking a Java implementation of a randomized version of quick-sort is plotted.



From the above graph we can conclude that for this particular example, the new search strategies PFS, BFPSS and RS, which take into account the probabilities, outperform the JPF's standard search strategies DFS and BFS. However, as we have shown in [3], the search strategies are in theory incomparable. That

is, for each pair of search strategies, there exists a probabilistic system such that the one strategy makes faster progress than the other. Also in practice we have seen that the search strategies are incomparable. However, for most probabilistic systems the new search strategies PFS, BFPSS and RS perform better than the JPF's original search strategies DFS and BFS.

5 Conclusion

We have extended JPF to a probabilistic model checker. In [3] we have introduced new search strategies that take into account the probabilities associated to the probabilistic choices of probabilistic systems. These strategies have been implemented in JPF. To measure the amount of progress of a probabilistic model checker has made, we have introduced a progress measure in [3]. JPF has been extended to keep track of this progress measure. We have shown that the new search strategies to make progress faster than JPF's original search strategies in most cases and, hence, are more appropriate to verify probabilistic systems.

Developing other search strategies, based on ideas from the field of scheduling, is a direction for future research. Our current extension of JPF can only track the progress of sequential code representing a probabilistic system. Being able to also handle concurrent code is another challenge for the future.

The development of a preliminary version of our tool helped us shaping the theory discussed in [3]. After having fully developed the theory, we re-implemented our tool. The current version of our tool[3] is better structured and more efficient.

References

1. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. International Journal on Software Tools for Technology Transfer **6**(2) (September 2004) 128–142
2. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2) (April 2003) 203–232
3. Zhang, X., van Breugel, F.: A progress measure for explicit-state probabilistic model-checkers. Submitted to the *25th Annual IEEE Symposium on Logic in Computer Science*. Available at `www.cse.yorku.ca/~franck/research/drafts/` (January 2010)
4. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM **42**(4) (July 1995) 857–907
5. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: Proceedings of the 6th International Conference on Quantitative Evaluation of Systems, Budapest, Hungary, IEEE (September 2009) 167–176

[3] Available at the URL `www.cse.yorku.ca/~franck/research/progress`.