Probabilistic Model Checking of Randomized Java Code*

Syyeda Zainab Fatmi[1], Xiang Chen[2], Yash Dhamija[1], Maeve Wildes[3], Qiyi Tang[4], and Franck van Breugel[1]

[1] York University, Toronto
[2] University of Waterloo
[3] McGill University, Montreal
[4] University of Oxford

Abstract. Java PathFinder (JPF) and PRISM are the most popular model checkers for Java code and systems that exhibit random behaviour, respectively. Our tools make it possible to use JPF and PRISM together. For the first time, probabilistic properties of randomized algorithms implemented in a modern programming language can be checked. Furthermore, our tools are accompanied by a large collection of randomized algorithms that we implemented in Java. From those Java applications and with the help of our tools, we have generated the largest collection of realistic labelled (discrete time) Markov chains.

Keywords: (probabilistic) model checking · Java PathFinder · PRISM

1 Introduction

Java PathFinder (JPF) is the most popular model checker for Java code. It takes as input Java bytecode and a configuration file. The latter includes which properties to check. As output, JPF produces a report that tells us, among other things, whether any of those properties have been violated. PRISM is the most popular probabilistic model checker. As input, it takes a model of a system that exhibits random behaviour. The model can be expressed in a simple language, but also as a labelled Markov chain (LMC). Furthermore, it takes a probabilistic property specified in a logic as input. PRISM checks, among other things, whether the model satisfies the property.

The popularity of these model checkers is reflected by the facts that JPF has been downloaded hundreds of times every month for almost two decades and PRISM has been downloaded more than 75,000 times[5]. The papers that describe JPF [77] and PRISM [48] have each been cited more than 1,800 times.

JPF is an explicit state model checker. It builds a model of the Java bytecode on the fly. This model can be seen as a directed graph, which is known as the state space. The vertices correspond to the states of JPF's virtual machine and the edges, also called transitions, represent sequences of bytecode instructions.

* Supported by the Natural Sciences and Engineering Research Council of Canada.
[5] www.prismmodelchecker.org/download.php

We have developed two extensions of JPF, jpf-label and jpf-probabilistic, that decorate the state space in orthogonal ways. jpf-label provides an easy way to label the states. To capture simple known facts about the states of JPF's virtual machine, jpf-label decorates those states with a set of atomic propositions. Our extension supports a range of atomic propositions. For example, it allows us to identify those states that are initial or final, those states in which a specific boolean static field is true, those states in which a specific method returns, etc. These atomic propositions may be used to express properties of the code. Such properties can be formalized in logics such as linear temporal logic (LTL) [60] and computation tree logic (CTL) [13].

Our extension jpf-probabilistic assigns probabilities to the transitions. Those probabilities reflect the random choices in the Java code. To make it easy for programmers to express those in Java and for our extension to detect them, we have introduced three Java classes. The class `Choice` contains the method `make` which takes an array of `double`s, say `p`, as argument. If $\sum_{0 \leq i < \texttt{p.length}} \texttt{p}[i] = 1.0$ then the method invocation `Choice.make(p)` returns i, where $0 \leq i < \texttt{p.length}$, with probability $\texttt{p}[i]$. For convenience, we also introduced the class `Coin` with the method `flip`. Furthermore, the method invocation `UniformChoice.make`(n) returns i, where $0 \leq i < n$, with probability $\frac{1}{n}$. By adding probabilities to the transitions, we turn the state space into a (discrete time) Markov chain (DTMC).

By default, JPF uses depth-first search (DFS) to traverse the state space. It also supports breadth-first search (BFS). jpf-probabilistic contains several search strategies that take the probabilities into account. In particular, it supports probability-first search [80], random search [80], softmax search [74], and ϵ-greedy search [74]. The latter two are inspired by reinforcement learning [73]. We discuss these new search strategies in more detail in Section 3. As we will see in Section 6, they are much more effective than DFS when model checking randomized Java code.

Our extensions jpf-label and jpf-probabilistic together with a converter allow us to use JPF and PRISM in tandem. Given a randomized algorithm implemented in Java and a configuration file, JPF with the help of jpf-label and jpf-probabilistic produces an LMC, that is, a Markov chain the states of which are labelled with atomic propositions. This chain can be graphically represented as a directed graph where the edges are labelled with probabilities and the vertices are coloured (where colours represent atomic propositions), and it can also be converted into a very similar LMC that can be fed into PRISM together with a probabilistic property. This provides the first model checking tool, depicted in Figure 1, that can check probabilistic properties of randomized algorithms implemented in a modern programming language.

Instead of using our tool, one could try to model randomized algorithms in PRISM's input language. However, numerous details of some algorithms cannot easily be handled directly by PRISM. Consider, for example, Frieze's randomized algorithm to find a Hamiltonian cycle in an undirected graph [24]. The algorithm uses lists which are rotated and reversed. Furthermore, the probabilities associated with the choices in the algorithm depend on the size of the lists and can

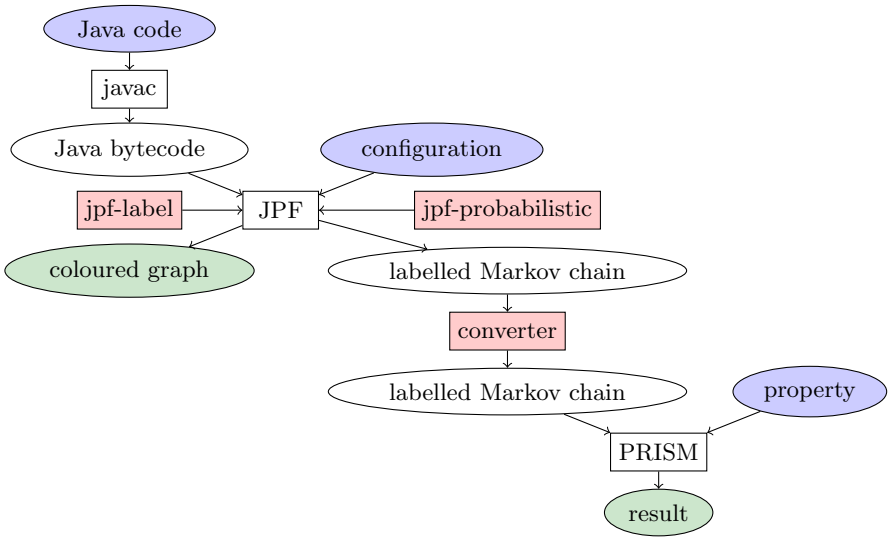

Fig. 1. The diagram provides an overview of the model checking tool. The ovals are data and the rectangles are tools. The blue ovals are input. The green ovals are output. The red rectangles are the parts that we developed.

be zero. Such features can easily be captured in Java but cannot be directly captured in PRISM's input language.

The Java implementations of sixty randomized algorithms are provided with our extension jpf-probabilistic. To illustrate how our tool can be used, we present three examples. In the first one, we consider the Miller-Rabin primality test [56,63]. This is a Monte Carlo algorithm as it may incorrectly report with small probability that the number provided as an argument is prime. This algorithm has been implemented in the method `isProbablePrime` of the class `java.math.BigInteger`. With our tool we can compute the probability of this algorithm erroneously reporting that a number is prime. We have applied our tool to several other Monte Carlo algorithms as well.

In the second example, we consider a variation of an algorithm due to Floyd and Rivest [21]. This is a Las Vegas algorithm as it always returns the correct result, however, the running time may vary. The algorithm selects the ith smallest of n numbers. Hence, it can be used to determine the median, which is an important clustering statistic (see, for example, [79]). Some steps of the algorithm may fail with small probability. If that happens, those steps need to be repeated. As a result, this algorithm gives rise to an infinite state space. Hence, JPF will eventually run out of memory when model checking a Java implementation of this algorithm. However, as we will show, with our tool we can compute the probability that the part of the state space that has not been fully explored by JPF is reached when the code is run. This provides us with a lower bound on the confidence in the verification results of JPF. For example, if JPF does not de-

tect any uncaught exceptions and the probability of the unexplored state space is 0.01, then it is guaranteed that an uncaught exception does not occur with at least probability 0.99. We have considered several other randomized algorithms that give rise to very large or infinite state spaces.

In the third example, we show that our tool can compute other quantitative properties of randomized algorithms implemented in Java. For example, we demonstrate that the probability that a randomly generated graph [18] is connected can be obtained by means of our tool.

As these examples will demonstrate, not only can PRISM provide quantitative information that enriches the qualitative verification results of JPF. PRISM can also turn a JPF out of memory error into valuable quantitative information about JPF's seemingly failed verification effort. Our extensions of JPF, jpf-label and jpf-probabilistic, as well as our converter are essential in both cases.

2 jpf-label

As we already mentioned, to express properties of Java code in terms of logics such as LTL and CTL, we need a way to specify atomic propositions and to label those states that satisfy them. For example, we may want to label the final states or those states in which a specific static boolean field is true. Currently, there is no model checker for Java code that labels states.

In the past, several extensions of JPF, all named jpf-ltl, have been developed that supported the checking of properties expressed in LTL.[6] None of these extensions is compatible with the latest version of JPF. In [14], Cuong and Cheng describe a tool that given a property expressed in LTL generates an extension of JPF that checks the property. Unfortunately, an implementation of such a tool is not available [12].

In the literature, the following categories of atomic propositions in the context of Java code are distinguished:

- static boolean fields and local boolean variables (jpf-ltl),
- boolean expressions built from static integer fields and local integer variables (jpf-ltl and [39]),
- method invocations (jpf-ltl and [14,39,72]),
- method returns and the values returned ([5,39]),
- thrown exceptions and the exception types ([39]), and
- AspectJ pointcuts ([72]).

Note that, apart from jpf-ltl and [14], all the above references describe verification tools that are not model checkers.

[6] Only one version of jpf-ltl is still available. This version is based on the algorithms described in [27] and can be found at the URL code.google.com/archive/p/jpf-ltl/source. Most of the code is more than 15 years old. JPF has changed a lot in the last 15 years, thus, this extension is incompatible with the current version of JPF.

Our extension jpf-label[7] implements the following 12 different ways to label states, including instances of all the above mentioned categories apart from the last one.

- `Initial`: labels the initial state.
- `End`: labels the final states, also known as end states in JPF.
- `BooleanStaticField`: labels states with the value of a static boolean field.
- `IntegerStaticField`: labels states with the value of a static integer field.
- `BooleanLocalVariable`: labels states with the value of a local boolean variable.
- `IntegerLocalVariable`: labels states with the value of a local integer variable.
- `InvokedMethod`: labels those states in which a method is invoked.
- `ReturnedBooleanMethod`: labels those states in which a method has returned with the boolean return value.
- `ReturnedIntegerMethod`: labels those states in which a method has returned with the integer return value.
- `ReturnedVoidMethod`: labels those states in which a void method has returned.
- `SynchronizedStaticMethod`: labels those states in which a synchronized static method acquires or has released the lock.
- `ThrownException`: labels those states in which an exception has been thrown.

The field, variable, method, or exception of interest is specified in the JPF configuration file. Our tool also allows users to easily implement their own state labelling by simply extending an abstract class containing only five methods.

Any code that JPF can handle, can also be handled by jpf-label. When we run JPF extended with jpf-label on Java code, it can generate a file that contains the state labelling. An example is provided in Figure 2. Furthermore, it can also produce a graphical representation of the labelled state space as a coloured directed graph, an example of which can be seen in Figure 9. The probabilities on the edges of this figure are produced by the extension jpf-probabilistic which we discuss in detail in the next section.

Our extension jpf-label consists of 2,693 lines of Java code. More details about the design and implementation of jpf-label can be found in [19].

3 jpf-probabilistic

Our extension jpf-probabilistic[8] decorates the transitions of the state space with probabilities, turning the state space into a DTMC. jpf-probabilistic can handle any Java code that can be model checked by JPF, which contains randomness, but does not contain any other sources of nondeterminism, such as concurrency.

[7] jpf-label is available at github.com/javapathfinder/jpf-label.
[8] jpf-probabilistic is available at github.com/javapathfinder/jpf-probabilistic.

```
0="init" 1="true__PrimalityTest_isPrime__II__Z" 2="end"
    ↪ 3="false__PrimalityTest_isPrime__II__Z"
-1: 0
2: 1
3: 2
4: 3
5: 2
6: 1
```

Fig. 2. The file `PrimalityTest.lab` contains the state labelling produced by JPF extended with our jpf-label from the Java app `PrimalityTest`. Line 1 lists the labels and their indices. The remaining lines provide the labelling for those states that have labels. For example, line 4 specifies that state 3 is labelled `"end"`, that is, it is a final state.

When we run JPF extended with jpf-probabilistic on randomized Java code, we can generate a file that contains the Markov chain corresponding to the code. An example can be found in Figure 3. When using both jpf-label and jpf-probabilistic, JPF can produce a graphical representation of the LMC as a directed graph where the vertices are coloured and the edges are labelled with probabilities (see Figure 9).

```
788962 1347606
-1 0 1.000000
0 1 0.200000
0 2 0.200000
0 3 0.200000
...
```

Fig. 3. The file `LazySelect.tra` contains the Markov chain produced by JPF extended with our jpf-probabilistic from the Java app `LazySelect`. The first line specifies the number of states and the number of transitions. The transitions and their probabilities are described in the remaining lines. Here we only show four transitions. Each transition is captured by its source state, its target state, and its probability. For example, line 5 specifies the transition from state 0 to state 3 with probability 0.200000.

By default, JPF uses depth-first search to traverse the state space. It also supports breadth-first search. Since our extension jpf-probabilitic associates probabilities with the transitions, these probabilities can be used to drive the search of the state space. Our extension provides four such search strategies.

Probability-first search (PFS), which was introduced by Zhang in [80], uses the probabilities of the transitions to select the next state to explore. In particular, it always chooses a state whose path along which it is discovered has the highest probability.

Random search (RS) [80] randomly selects a state among the states that have been discovered, but that have not yet been fully explored. The chance of choosing a state is proportional to the probability of the path along which the state has been discovered. Let us make that precise. Assume that $\{s_0, \ldots, s_n\}$ is the set of states that have been discovered but their outgoing transitions have not all been explored yet. Then RS chooses state s_j with probability

$$\frac{p(s_j)}{\sum_{0 \leq i \leq n} p(s_i)},$$

where $p(s_i)$ is the probability of the path along which s_i is discovered.

In [74], Tang introduced two search strategies inspired by reinforcement learning [73]. The softmax search (SMS) selects the next state according to a Gibbs distribution. Assume again that $\{s_0, \ldots, s_n\}$ is the set of states that have been discovered but not yet fully explored. Then SMS chooses state s_j with probability

$$\frac{e^{p(s_j)/\tau}}{\sum_{0 \leq i \leq n} e^{p(s_i)/\tau}},$$

where $p(s_i)$ is the probability of the path along which s_i is discovered and the constant τ is called the temperature. This constant should be a positive real number. The ϵ-greedy search (EGS) relies on a parameter $\epsilon \in (0, 1)$. It combines RS and PFS in such a way that with probability $1 - \epsilon$ it behaves like PFS and with probability ϵ it behaves like RS. These different search strategies often visit the states in a different order and, as a result, may visit different parts of the state space and, hence, detect bugs in different parts of the code (see Section 6).

An earlier version of jpf-probabilistic has been discussed in [81]. Since then, a lot has changed. For example, the search strategies SMS and EGS have been added and the search strategies PFS and RS have been implemented more efficiently. Also, the ability of jpf-probabilistic to generate a file that contains the Markov chain and to produce a graphical representation of the LMC are both new. Furthermore, numerous examples have been added (see Appendix A). The current version of jpf-probabilistic contains 14,224 lines of Java code. Only 996 lines of Java code of the original version of jpf-probabilistic have remained and the other 573 lines of Java code have been deleted or replaced.

4 Our Converter

The format of the transition and labelling files generated by JPF differs slightly from PRISM's input format. Whereas JPF numbers its states starting from -1, PRISM starts at zero. JPF may produce multiple transitions between a given pair of states, while PRISM allows at most one transition between any pair of states. Furthermore, in PRISM a label may only consist of letters, digits and the underscore character, and it can neither begin with a digit nor contain any whitespace. Additionally, a label should not be a reserved keyword in PRISM. PRISM also requires that the initial states of the model are labelled as such.

Therefore, we have implemented a simple converter, named `JPFtoPRISM`, that renumbers the states in the transition and label files. The converter also checks if all labels satisfy the above mentioned restrictions. Furthermore, if the initial state of the model is not labelled, the converter adds the label `"init"` to the initial state. If JPF has produced multiple transitions from a given source state to a given target state, then the converter collapses those transitions into a single transition between the source and target state by adding up the transition probabilities. Moreover, if the probabilities of the outgoing transitions of each state do not sum to one, the converter adds a labelled sink state for the remaining probability. This ensures that if JPF has not traversed the state space completely, for example, because it ran out of memory, then the resulting LMC's transition matrix is a right stochastic matrix, preventing a deadlock warning in PRISM.

We expand the explanation of the translation done by the converter `JPFtoPRISM` through a small example. Consider the labelled Markov chain represented by the labelling file shown in Figure 4 and the transition file shown in Figure 5.

```
1 0="end"
2 2: 0
```

Fig. 4. Line 1 lists the labels and their indices. In this case, there is only one label, namely `"end"`, with index 0. Line 2 specifies that state 2 is labelled `"end"`, that is, it is a final state.

```
1 4 5
2 -1 0 1.0
3 0 1 0.25
4 0 1 0.25
5 0 2 0.25
6 2 2 1.0
```

Fig. 5. The first line specifies that there are four states and five transitions. The five transitions and their probabilities are described in the remaining lines. Each transition is captured by its source state, its target state, and its probability.

Using our converter, the LMC described above is transformed into an LMC in PRISM's format, represented by the labelling file shown in Figure 6 and the transition file shown in Figure 7. The converter renumbers the states such that the numbering begins at 0. Since the initial state of the model is not labelled, the converter adds the label `"init"` to the initial state. The converter also collapses multiple transitions from a given source state to a given target state into a single

transition by adding up the transition probabilities. Finally, the converter adds a sink state for the remaining probability of those states of which the probabilities of their outgoing transitions do not sum to one.

```
1 0="end" 1="init" 2="sink"
2 0: 1
3 3: 0
4 4: 2
```

Fig. 6. Line 1 lists the labels and their indices. Two labels have been added, namely `"init"` and `"sink"`, to label the initial state 0 and sink state 4, respectively. The states have been renumbered. Note that line 3 specifies that the final state is now state 3.

```
1 5 7
2 0 1 1.000000
3 1 2 0.500000
4 1 3 0.250000
5 1 4 0.250000
6 2 4 1.000000
7 3 3 1.000000
8 4 4 1.000000
```

Fig. 7. The first line specifies that there are five states and seven transitions. The states have been renumbered. The two transitions from state 0 to state 1 in Figure 5 have been combined into one transition from state 1 to state 2. Transitions to the sink state, state 4, have been added from those states that have not yet been fully explored (state 1 and 2 in the transformed system). Note that the sink state transitions to itself with probability 1.0.

5 Monte Carlo Algorithms

Monte Carlo algorithms are randomized algorithms that may produce incorrect results with a small probability. As we will show, JPF and our extensions jpf-label and jpf-probabilistic together with our converter and PRISM can compute the probability that a Monte Carlo algorithm implemented in Java gets it wrong.

Numerous of the algorithms listed in Appendix A are Monte Carlo algorithms, including the primality tests due to (1) Fermat [55], (2) Lucas, (3) Miller and Rabin [56,63], and (4) Solovay and Strassen [70]. These algorithms determine whether a number given as input is prime. The algorithms may erroneously report that the input number is prime. As most Monte Carlo algorithms, the

algorithms contain a main loop. The more iterations of this loop, also known as trials, are executed, the lower the probability that the algorithms return an incorrect result.

We have implemented the Miller-Rabin primality test in Java in a class called `MillerRabinPrimalityTest`, abbreviated to `PrimalityTest` below. The randomization in the code is captured by jpf-probabilistic's `UniformChoice.make` method. We configure JPF as specified in Figure 8. Running JPF with this configuration file results in the creation of the file named `PrimalityTest.dot` in DOT format. The resulting coloured graph is depicted in Figure 9.

```
1  target = PrimalityTest
2  target.args = 9,2
3  classpath = <directory containing PrimalityTest.class>
4
5  @using jpf-label
6  label.class = label.Initial; label.End; label.ReturnedBooleanMethod
7  label.ReturnedBooleanMethod.method = PrimalityTest.isPrime(int,int)
8
9  @using jpf-probabilistic
10 listener = probabilistic.listener.StateSpaceDot,
       ↪ probabilistic.listener.StateLabelVisitor
11 probabilistic.listener.StateSpaceDot.precision = 3
```

Fig. 8. This JPF configuration file specifies that the Java app named `PrimalityTest` with the command line arguments `9` (the number to be tested for primality) and `2` (the number of trials) is to be model checked by JPF. The `classpath` tells JPF where to find the bytecode of the app (alike an ordinary Java virtual machine's classpath). Line 5 and 8 specify that our extensions jpf-label and jpf-probabilistic are used. Line 6 specifies that the initial state and the final states (also known as end states in JPF) should be labelled, as well as those states in which the method `isPrime` and the class `PrimalityTest` that takes two `int`s as arguments (as specified in line 7) returns. Finally, line 10 specifies that JPF should generate a graphical representation of the state space (which forms an LMC) and line 11 captures that the probabilities of the transitions should be depicted with three digits precision.

The Miller-Rabin primality test is correct when a prime number is provided as input. We compute the probability that the algorithm returns the wrong result when a composite number is provided as input. We first run JPF as described above, but using this time `probabilistic.listener.StateSpaceText` and `label.StateLabelText` in line 10 of Figure 8 instead. As a result, JPF creates the file `PrimalityTest.tra` that contains the transitions and their probabilities, and the file `PrimalityTest.lab` that contains the state labelling. Together they specify an LMC. The label `"true__PrimalityTest_isPrime__II__Z"`, abbreviated below as `"incorrect"`, captures that the method `isPrime` of the class `PrimalityTest`, which takes two arguments of type `int` and returns a value of

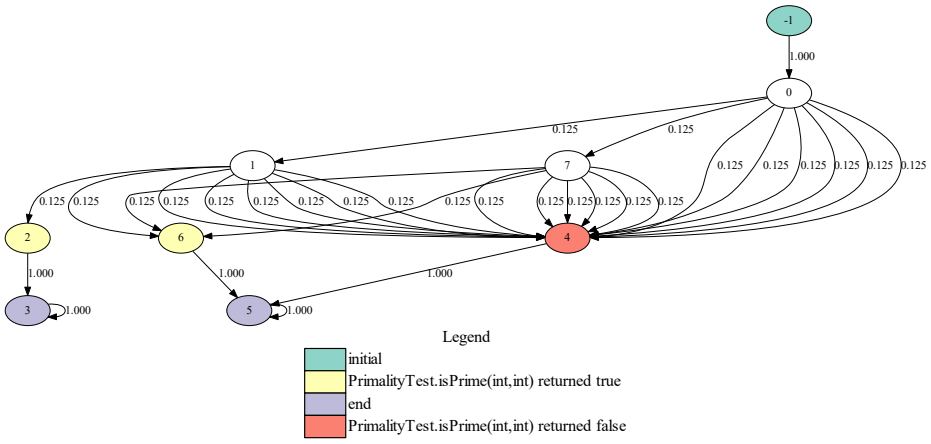

Fig. 9. This coloured graph has been generated by JPF extended with jpf-label and jpf-probabilistic. It represents the state space for the Miller-Rabin primality test run for two trials for the input number 9. The initial state (state -1) and the final states (states 3 and 5) are labelled, as well as those states in which the static method that determines whether the number is prime returns true (states 2 and 6) and false (state 4).

type `boolean`, returns the value `true`. This label captures the scenario in which the method returns true but the input is not a prime. Note that we use name mangling similar to that used in the Java native interface [51].

Subsequently, we use our converter to transform the LMC into PRISM's format. Finally, we use PRISM to compute for this LMC the property `P=? [ F "incorrect" ]`. That is, PRISM computes the probability that the LTL property `F "incorrect"` holds. This property specifies that eventually a state labelled `"incorrect"`, that is, a state in which the method `isPrime` of the class `PrimalityTest` returns `true`, is reached. PRISM returns the probability 0.0625, which corresponds to reaching either state 2 or state 6 in Figure 9.

6 Very Large and Infinite State Spaces

The size of the underlying LMC is often too large for JPF to explore entirely, before running out of time or memory. In such a case, we can measure the amount of progress made by JPF, using our extension jpf-probabilistic and our converter together with PRISM, as we will show below.

The lazy select algorithm [21] selects the ith smallest of n numbers. Some steps of the algorithm may fail with small probability. If that happens, those steps need to be repeated. As a result, this algorithm gives rise to an infinite state space. We implemented the algorithm in Java, again using jpf-probabilistic's `UniformChoice.make` method to capture randomization.

When we use JPF in combination with our extension jpf-probabilistic, to model check the Java code to select the third smallest of five elements, JPF runs

out of its 10 GB of memory after 2 minutes and 9 seconds. In that time, JPF visits 788,962 states and does not detect any violations of properties such as uncaught exceptions. However, since JPF does not completely traverse the infinite state space, its verification effort provides very little, if any, useful information.

By using PRISM in combination with JPF, we can extract useful quantitative information from a seemingly failed verification effort. This is accomplished as follows. Instead of letting JPF run out of memory, JPF can be configured so that it stops just before running out of memory. Our extensions jpf-label and jpf-probabilistic generate the LMC. Subsequently, this LMC is converted into PRISM's format by means of our `JPFtoPRISM`. Since not all states have been fully explored, the converter also adds a sink state to the LMC as well as a transition to this sink state from all states that have not been fully explored by JPF and also labels the sink state. Finally, we use PRISM to determine the probability that the sink state is eventually reached by computing the property `P=? [ F "sink" ]`. For the above mentioned LMC with 788,962 states, this property has a value less than 0.00004. As a consequence, with more than probability 0.99996 only fully explored states are reached. Hence, if we run the Java code then with at least probability 0.99996 we will not encounter any violation of the properties checked by JPF. This number represents the progress made by JPF [82].

JPF provides two search strategies: DFS and BFS. As mentioned in Section 3, jpf-probabilistic provides a number of other search strategies that take the probabilities into account. Since different search strategies may visit states in different orders, they may make progress at different rates. As shown in Figure 10, this is indeed the case for the Java implementation of lazy select. Some of the search strategies that take the probabilities into account make more progress than BFS. DFS, JPF's default search strategy, makes no progress for this particular example.

7 Other Quantitative Properties

In addition to determining the probability that a Monte Carlo algorithm returns an incorrect result and the progress made by JPF on a large or infinite state space, our tool can check a wide range of other quantitative properties of randomized algorithms implemented in Java.

The Erdös-Rényi model [18] is a model for generating random graphs. In this model, a graph with a given number of vertices is constructed by placing an edge between each pair of vertices with a given probability, independent from every other edge. We implemented a version of the algorithm to generate random undirected graphs in Java in the class `ErdosRenyiUndirectedModel`. We use jpf-probabilistic's `Choice.make` method to express the random choices in the code.

Assume that we would like to determine the probability that the graph generated by the algorithm is connected. We add a boolean method to our class, called `isConnected`, that returns true if the graph is connected and false otherwise. We run JPF with the configurations specified in Figure 11, which results

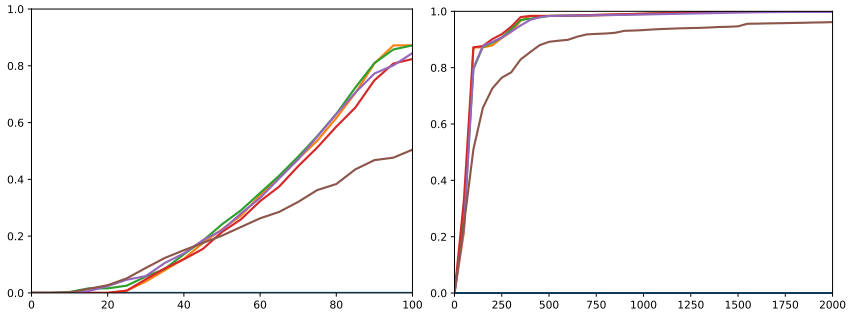

Fig. 10. This graph depicts results of the model checking tool applied to the Java code implementing lazy select that selects the third smallest of five elements. The x-axis represents time in milliseconds. The y-axis represents the progress made by JPF. The colours represent the different search strategies: ● = depth-first search, ● = breadth-first search, ● = ϵ-greedy search, ● = probability-first search, ● = random search, ● = softmax search. The graph on the left zooms in on the first 100 milliseconds. The progress of depth-first search is zero and, therefore, coincides with the x-axis.

```
1  target = ErdosRenyiUndirectedModel
2  target.args = 5,0.6
3  classpath = <directory containing ErdosRenyiUndirectedModel.class>
4
5  @using jpf-label
6  label.class = label.ReturnedBooleanMethod
7  label.ReturnedBooleanMethod.method =
       ↪ ErdosRenyiUndirectedModel.isConnected()
8
9  @using jpf-probabilistic
10 listener = probabilistic.listener.StateSpaceText,label.StateLabelText
```

Fig. 11. This JPF configuration file specifies that the Java app named `ErdosRenyiUndirectedModel` with the command line arguments `5` (the number of vertices in the graph) and `0.6` (the probability of adding an edge between two vertices) is to be model checked by JPF. Line 5 and 8 specify that our extensions jpf-label and jpf-probabilistic are used. Line 6 and 7 specify that those states in which the boolean method `isConnected` in the class `ErdosRenyiUndirectedModel` returns should be labelled. Line 10 specifies that JPF should generate a textual representation of the state space.

in the creation of a transition and labelling file that represent the underlying LMC.

Using our converter `JPFtoPRISM`, we transform the LMC produced by JPF into PRISM's format. We then run PRISM to compute the property `P=? [ F "true__ErdosRenyiUndirectedModel_isConnected____Z" ]`, which captures the probability that eventually a state is reached in which the boolean

method `isConnected` of the class `ErdosRenyiUndirectedModel` returns `true`. By varying the number of vertices in the random graph and the probability of placing an edge between any two vertices, we construct the graph shown in Figure 12.

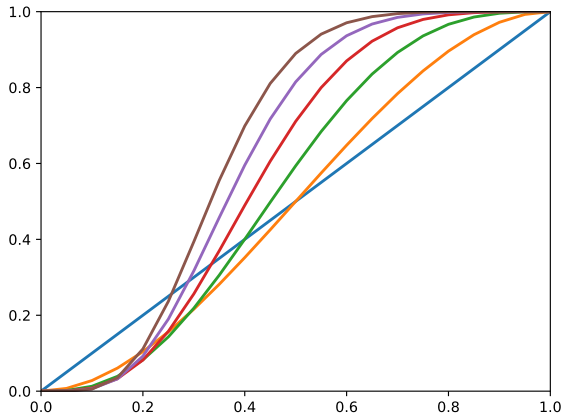

Fig. 12. This graph depicts results of the model checking tool applied to the Java code implementing the Erdös-Rényi model. The x-axis represents the probability of adding an edge between two vertices. The y-axis represents the probability that the generated graph is connected. The colours represent the number of vertices in the generated graph: ● = 2, ● = 3, ● = 4, ● = 5, ● = 6, ● = 7

8 Overhead

We monitored the memory and time usage of our tool on the examples presented in Appendix A. In all cases we have observed so far, the overhead of jpf-label and jpf-probabilistic is very limited.

Consider the algorithm to determine whether an integer array given as input has a majority element [57]. The algorithm is a Monte Carlo algorithm and may erroneously report that the given array does not have a majority element. The algorithm contains a main loop. The more iterations of this loop, also known as trials, are executed, the lower the probability that the algorithm returns an incorrect result.

We have implemented this algorithm in Java in a class called `HasMajorityElement`. We provide as input an integer array of size eleven. By increasing the number of trials we can increase the size of the state space linearly.

The amounts of time (in seconds) used by JPF without and with jpf-label and jpf-probabilistic are shown in Figure 13. jpf-probabilistic has virtually no overhead and jpf-label increases the time used by JPF by a factor of approximately 1.2. The amounts of heap memory (in MB) used by JPF without and

with jpf-label and jpf-probabilistic are shown in Figure 14. For both jpf-label and jpf-probabilistic, the difference is only a few MB.

It should be mentioned that one can easily write a Java application for which the memory overhead caused by jpf-label is arbitrarily large (for example, see Figure 15). However, such a Java application, which has been designed specifically to cause jpf-label to create enormous amounts of extra states, one does not encounter in practice.

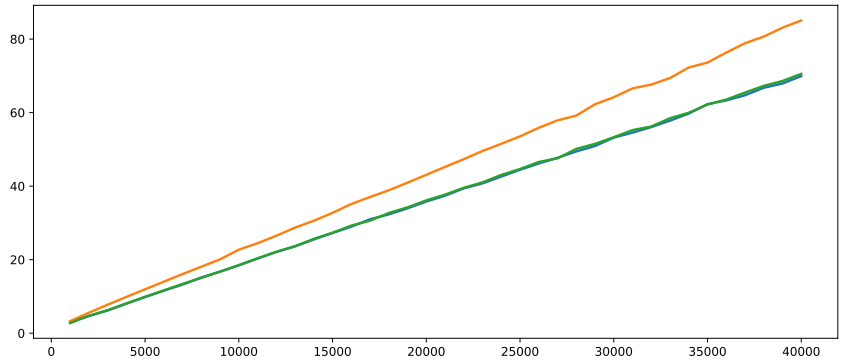

Fig. 13. This graph depicts the time used by JPF applied to the Java code implementing the majority element algorithm. The x-axis represents the number of iterations of the main loop. The y-axis represents the amount of time in seconds. The colours represent the following configurations: ● = JPF run without jpf-label and jpf-probabilistic, ● = JPF run with jpf-label, ● = JPF run with jpf-probabilistic.

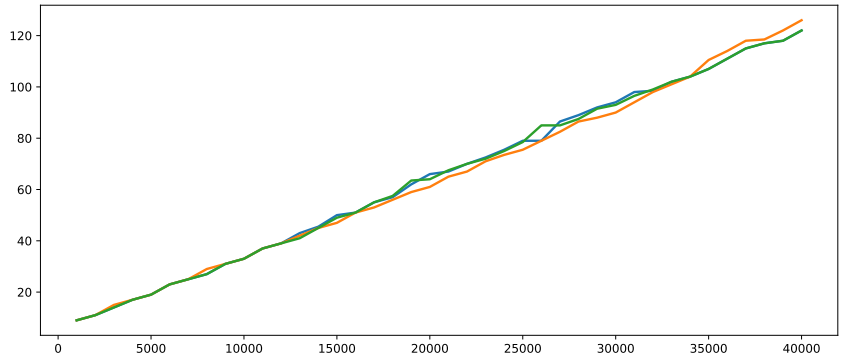

Fig. 14. This graph depicts the memory usage of JPF applied to the Java code implementing the majority element algorithm. The x-axis represents the number of iterations of the main loop. The y-axis represents the amount of heap memory in MB. The colours represent the following configurations: ● = JPF run without jpf-label and jpf-probabilistic, ● = JPF run with jpf-label, ● = JPF run with jpf-probabilistic.

```
1  public class Example {
2    public static boolean flip = true;
3    public static void main(String[] args) {
4      int n = Integer.parseInt(args[0]);
5      for (int i = 0; i < n; i++) {
6        flip = false;
7        flip = true;
8      }
9    }
10 }
```

Fig. 15. Running this example without jpf-label results in a single state. Running the code with jpf-label, while labelling the states with the value of the field `flip`, and passing the value `n` as command-line argument results in `2n+2` states. However, this is not a realistic example.

9 Conclusion

Our extensions of JPF, jpf-label and jpf-probabilistic, expand the functionality of the model checker. The former provides an easy way to the label the states and the latter assigns probabilities to the transitions and introduces new search strategies. Both extensions have been designed in such a way that they themselves can be easily extended.

Our extensions together with our converter `JPFtoPRISM` build a bridge between the model checkers JPF and PRISM. They allow us to use them in tandem. For example, we now can check properties expressed in logics such as LTL [60] and PCTL [31] of randomized Java code. Furthermore, we can use PRISM to supplement JPF's qualitative results with quantitative information.

To determine their performance, many probabilistic model checking algorithms are run on randomly generated LMCs. Since these algorithms are applied in practice to LMCs that are far from random, there is a pressing need for realistic LMCs. From the Java implementations of randomized algorithms that accompany jpf-probabilistic we can generate a large collection of LMCs. It almost doubles the number of available realistic LMCs in PRISM's collection. For all of these examples, the overhead of jpf-label and jpf-probabilistic is very limited, as discussed in Section 8.

Our tool handles any Java (byte)code acceptable by JPF (currently JPF fully supports Java 8 and most features of Java 11) that does not contain other forms of nondeterminism, such as concurrency, because such Java (byte)code gives rise to a probabilistic automaton [67] instead of a DTMC. Extending the tool so that it can handle other forms of nondeterminism is left for future research.

References

1. Leonard Adleman, Kenneth Manders, and Gary L. Miller. On taking roots in finite fields. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 175–178, Providence, RI, USA, October/November 1977. IEEE.
2. Noga Alon, Manuel Blum, Amos Fiat, Sampath Kannan, Moni Naor, and Rafail Ostrovsky. Matching nuts and bolts. In Daniel D. Sleator, editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 690–696, Arlington, VA, USA, January 1994. ACM/SIAM.
3. Noga Alon, Zvi Galil, Oded Margalit, and Moni Naor. Witnesses for Boolean matrix multiplication and for shortest paths. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 417–427, Pittsburgh, PA, USA, October 1992. IEEE.
4. Musab AlTurki and José Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Proceedings of the 4th International Conference on Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392, Winchester, UK, August/September 2011. Springer-Verlag.
5. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Online testing of LTL properties for Java code. In Valeria Bertacco and Axel Legay, editors, *Proceedings of the 9th International Haifa Verification Conference*, volume 8244 of *Lecture Notes in Computer Science*, pages 95–111, Haifa, Israel, November 2013. Springer-Verlag.
6. Howard Barringer. Randomized algorithms - a brief introduction. 2010.
7. Elwyn R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24(111):713–735, July 1970.
8. Allan Borodin, Stephen A. Cook, Patrick W. Dymond, Walter L. Ruzzo, and Martin Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18(3):559–578, June 1989.
9. Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro D'Argenio, editors, *Proceedings of the 10th International Conference on Quantitative Evaluation of Systems*, volume 8054 of *Lecture Notes in Computer Science*, pages 160–164, Buenos Aires, Argentina, August 2013. Springer-Verlag.
10. Andrei Z. Broder. On the resemblance and containment of documents. In Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro, and James A. Storer, editors, *Proceedings of the International Conference on Compression and Complexity of SEQUENCES*, pages 21–29, Positano, Salerno, Italy, June 1997. IEEE.
11. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):14–154, April 1979.
12. Khoo Siau Cheng. Personal communication, October 2019.
13. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, NY, USA, May 1981. Springer-Verlag.
14. Nguyen Anh Cuong and Khoo Siau Cheng. Towards automation of LTL verification for Java Pathfinder. In *Proceedings of the 15th National Undergraduate Reseach Opportunities Programme Congress*, Singapore, March 2010. National University of Singapore.

15. Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is coming: A modern probabilistic model checker. In Rupak Majumdar and Viktor Kuncak, editors, *Proceedings of the 29th International Conference on Computer Aided Verification*, volume 10427 of *Lecture Notes in Computer Science*, pages 592–600, Heidelberg, Germany, July 2017. Springer-Verlag.
16. Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *International Journal on Software Tools for Technology Transfer*, 6(4):320–341, August 2004.
17. Marco Dorigo and Luca M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.
18. Paul Erdös and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae*, 6:290–297, 1959.
19. Syyeda Zainab Fatmi. Probabilistic model checking of randomized Java code. Master's thesis, York University, Toronto, Canada, 2020. Draft available at www.eecs.yorku.ca/~zfatmi13/thesis.pdf.
20. Ronald A. Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Oliver & Boyd, London, UK, 4th edition, 1953.
21. Robert Floyd and Ronald Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, March 1975.
22. W. Donald Frazer and Archie C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, July 1970.
23. Rūsiņš Mārtiņš Freivalds. Probabilistic machines can use less running time. In Bruce Gilchrist, editor, *Proceedings of the 7th IFIP Congress on Information Processing*, pages 839–842, Toronto, Canada, August 1977. North-Holland.
24. Alan M. Frieze. Finding hamilton cycles in sparse random graphs. *Journal of Combinatorial Theory, Series B*, 44(2):230–250, April 1988.
25. David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–14, January 1962.
26. Pawel Gawrychowski, Jukka Suomela, and Przemyslaw Uznanski. Randomized algorithms for finding a majority element. In Rasmus Pagh, editor, *Proceedings of the 15th Scandinavian Symposium and Workshops on Algorithm Theory*, volume 53 of *Leibniz International Proceedings in Informatics*, pages 9:1–9:14, Reykjavik, Iceland, June 2016. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
27. Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of ltl formulae to büchi automata. In Doron A. Peled and Moshe Y. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, TX, USA, November 2002. Springer-Verlag.
28. Ernst Moritz Hahn, Tingting Han, and Lijun Zhang. Synthesis for PCTL in parametric Markov decision processes. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *Proceedings of the 3rd International Symposium on NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 146–161, Pasadena, CA, USA, April 2011. Springer-Verlag.
29. Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. INFAMY: An infinite-state Markov model checker. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 641–647, Grenoble, France, June/July 2009. Springer-Verlag.

30. Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. iscasMc: A web-based probabilistic model checker. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *Proceedings of the 19th International Symposium on Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 312–317, Singapore, May 2014. Springer-Verlag.
31. Hans Hansson and Bengt Jonsson. A framework for reasoning about time and reliability. In *Proceedings of the 10th Real-Time Systems Symposium*, pages 102–111, Santa Monica, CA, USA, December 1989. IEEE.
32. Arnd Hartmanns and Holger Hermanns. The Modest toolset: An integrated environment for quantitative modelling and verification. In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598, Grenoble, France, April 2014. Springer-Verlag.
33. Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The quantitative verification benchmark set. In Tomás Vojnar and Lijun Zhang, editors, *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 11427 of *Lecture Notes in Computer Science*, pages 344–350, Prague, Czech Republic, April 2019. Springer-Verlag.
34. Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84, Venice, Italy, January 2004. Springer-Verlag.
35. Charles A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.
36. Charles A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, July 1961.
37. Norman L. Johnson and Samuel Kotz. *Urn models and their application: An approach to modern discrete probability theory*. Wiley, New York, NY, USA, 1977.
38. Guy L. Steele Jr. and Jean-Baptiste Tristan. Adding approximate counters. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 15:1–15:12, Barcelona, Spain, March 2016. ACM.
39. Kari Kähkönen, Jani Lampinen, Keijo Heljanko, and Ilkka Niemelä. The LIME interface specification language and runtime monitoring tool. In Saddek Bensalem and Doron Peled, editors, *Proceedings of the 9th International Workshop on Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 93–100, Grenoble, France, June 2009. Springer-Verlag.
40. David R. Karger and Rajeev Motwani. Derandomization through approximation: An NC algorithm for minimum cuts. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 497–506, Montreal, Quebec, Canada, May 1994. ACM.
41. Narendra Karmarkar, Richard M. Karp, Richard J. Lipton, László Lovász, and Michael Luby. A Monte-Carlo algorithm for estimating the permanent. *SIAM Journal on Computing*, 22(2):284–293, April 1993.
42. Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1-3):165–201, November 1991.

43. Richard M. Karp, Michael Luby, and Neal Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, September 1989.
44. Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.
45. Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6(1):35–48, March 1986.
46. Joost-Pieter Katoen, Ivan Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90–104, February 2011.
47. Donald Knuth and Andrew Yao. The complexity of nonuniform random number generation. In Joseph Traub, editor, *Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity*, pages 375–428, Pittsburgh, PA, USA, April 1976. Academic Press.
48. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591, Snowbird, UT, USA, July 2011. Springer-Verlag.
49. YoungMin Kwon and Gul Agha. Linear inequality LTL (iLTL): A model checker for discrete time Markov chains. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Proceedings of the 6th International Conference on Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 194–208, Seattle, WA, USA, November 2004. Springer-Verlag.
50. Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
51. Sheng Liang. *The Java native interface: Programmer's guide and specification*. Addison-Wesley, Reading, MA, USA, 1999.
52. Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 1–10, Providence, RI, USA, May 1985. ACM.
53. Michael Luby. Removing randomness in parallel computation without a processor penalty. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 162–173, White Plains, NY, USA, October 1988. IEEE.
54. Colin McDiarmid. A random recolouring method for graphs and hypergraphs. *Combinatorics, Probability and Computing*, 2(3):363–365, September 1993.
55. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, Baco Raton, FL, USA, 1997.
56. Gary L. Miller. Riemann's hypothesis and tests for primality. In William C. Rounds, Nancy Martin, Jack W. Carlyle, and Michael A. Harrison, editors, *Proceedings of the 7th Annual ACM Symposium on Theory of Computing*, pages 234–239, Albuquerque, NM, USA, May 1975. ACM.
57. Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.
58. Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, page 345–354, New York, NY, USA, January 1987. ACM.

59. Christos H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 163–169, San Juan, Puerto Rico, October 1991. IEEE.
60. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, USA, October/November 1977. IEEE.
61. John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, September 1975.
62. William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.
63. Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, February 1980.
64. Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, Cambridge, MA, USA, 1981.
65. Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, December 1987.
66. Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, October 1980.
67. Roberto Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1995.
68. Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):540–545, October 1996.
69. Marc Snir. Lower bounds on probabilistic linear decision trees. *Theoretical Computer Science*, 38:69–82, 1985.
70. Robert M. Solovay and Volker Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, March 1977.
71. Robert H. Morris Sr. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, October 1978.
72. Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In Howard Barringer, Bernd Finkbeiner, Yuri Gurevich, and Henny Sipma, editors, *Proceedings of the 5th Workshop on Runtime Verification*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 109–124, Edinburgh, Scotland, July 2005. Elsevier.
73. Richard Sutton and Andrew Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, USA, 2018.
74. Qiyi Tang. Guiding probabilistic model checkers by reinforcement learning. Master's thesis, University of Oxford, Oxford, UK, September 2013.
75. Leslie G. Valiant and Gordon J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, Milwaukee, WI, USA, May 1981. ACM.
76. Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin Germany, 2003.
77. Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
78. John von Neumann. Various techniques used in connection with random digits. In A. S. Householder, G. E. Forsythe, and H. H. Germond, editors, *Monte Carlo*

Method, volume 12 of *National Bureau of Standards Applied Mathematics Series*, chapter 13, pages 36–38. US Government Printing Office, Washington, DC, USA, 1951.

79. Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, June 2015.
80. Xin Zhang. Checking progress of model checking randomized algorithms. Master's thesis, York University, Toronto, Canada, July 2010.
81. Xin Zhang and Franck van Breugel. Model checking randomized algorithms with Java PathFinder. In *Proceedings of the 7th International Conference on the Quantitative Evaluation of Systems*, pages 157–158, Williamsburg, VA, USA, September 2010. IEEE.
82. Xin Zhang and Franck van Breugel. A progress measure for explicit-state probabilistic model-checkers. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Proceedings of the 38th International Colloquium on Automata, Languages and Programming*, volume 6756 of *Lecture Notes in Computer Science*, pages 283–294, Zurich, Switzerland, July 2011. Springer-Verlag.
83. Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226, Marseille, France, June 1979. Springer-Verlag.

A Examples

Our extension jpf-probabilistic includes Java implementations of the following randomized algorithms, all but two different from those provided by PRISM, on which we have applied our tool. Most of the algorithms in our collection have parameters.

1. `AddingApproximateCounters`: adds two approximate counters [38]
2. `AntColonySystem`: finds a near-optimal solution to the traveling salesman problem [17]
3. `ApproximateCardinality`: estimates the cardinality of the union of the given collection of sets [43]
4. `ApproximateCounter`: counts approximately a large number of events using a small amount of memory [71]
5. `ApproximateIntegral`: (approximately) integrates a function [6]
6. `ApproximatePI`: approximates π
7. `ApproximatePermanent`: approximates the permanent of a 0-1 matrix [41]
8. `BooleanProductWitnessMatrix`: given two $n \times n$ boolean matrices A and B, finds a witness matrix W for $P = AB$, such that each entry W_{ij} is a witness for P_{ij}, if any, and 0 if there is no witness (a witness for P_{ij} is an index $k \in \{1, \ldots, n\}$ such that $A_{ik} \wedge B_{kj}$) [3]
9. `ByzantineAgreement`: processors, some of which may be faulty, come to an agreement using the Byzantine agreement protocol [50]
10. `DirectedReachability`: randomly walks through a directed graph from a source vertex in search for a target vertex [8]
11. `ErdosRenyiDirectedModel`: creates a random directed graph [18]
12. `ErdosRenyiUndirectedModel`: creates a random undirected graph [18]
13. `FairBaisedCoin`: makes a fair coin from a biased coin [78]
14. `FermatPrimalityTest`: determines if a number is prime using Fermat's approach [55]
15. `FisherYatesShuffle`: generates a random permutation of the elements of a list [20]
16. `FreivaldsTechnique`: verifies the result of a matrix multiplication [23]
17. `GaleShapleyStableMarriage`: verifies that the proposal algorithm solves the stable marriage problem [25]
18. `GameTree`: evaluates a game tree [69]
19. `GraphPartition`: partitions the vertices of an undirected graph into two sets so that at least half of the edges have one end point in the one set and one end point in the other set [53]
20. `GroupGenerator`: finds a generator of the group $\mathbb{Z}_p$ under multiplication [57]
21. `HamiltonianCycle`: finds a Hamiltonian cycle in an undirected graph [24]
22. `HasMajorityElement`: determines if an integer array has a majority element [57]
23. `HasPerfectMatching`: determines if a bipartite graph has a perfect matching [45]
24. `HashTable`: implements a hash table using a random hash function from a 2-universal hash family [11]

25. `HypergraphColouring`: finds a 2-colouring for a hypergraph [54]
26. `IndependentSet`: finds an independent set from a graph [52]
27. `KargersMinimumCut`: finds the size of a minimum cut of a undirected graph [57]
28. `KnuthYaoDie`: implements a die by means of a coin [47]
29. `LatticeApproximation`: given $n \times n$-matrix of integers A and a n-vector of reals p, finds a n-vector of reals q minimizing $\|A(p-q)\|_\infty$ [57]
30. `LazySelect`: selects the kth-smallest element of a set [21]
31. `LucasPrimalityTest`: determines if a number is prime using the Lucas primality test
32. `MajorityElement`: finds the majority element in a list [26]
33. `MatchingNutsAndBolts`: rearranges a collections of nuts and bolts of different widths such that matching nuts and bolts have the same index [2]
34. `MillerRabinPrimalityTest`: determines if a number is prime using the Miller-Rabin primality test [56,63]
35. `MinHash`: estimates how similar two sets are by approximating the Jaccard index between the two sets [10]
36. `PerfectMatching`: finds a perfect matching in a graph with at least one perfect matching [58]
37. `PollardsIntegerFactorization`: finds a factor of an integer [61]
38. `PolyasUrn`: picks random balls out of an urn and replaces them by balls of the same colour [37]
39. `PolynomialIdentities`: applies Frievald's technique to the verification of identities involving polynomials [23,83,66]
40. `QuadraticResidue`: finds the square roots of an integer in a field $\mathbb{Z}_p$ [1]
41. `Queens`: attempts to place a queen on each row of an $n \times n$ chess board such that no queen can attack another [6]
42. `QuickSelect`: finds the k-th smallest element from a collection [36]
43. `QuickSort`: sorts a list [35]
44. `RabinKarpPatternMatching`: finds the first occurrence (if any) of a given pattern in a string [44]
45. `RabinsFingerprint`: verifies the equality of strings using fingerprints [64]
46. `RandomizedBinarySearch`: a Las Vegas randomized binary search algorithm
47. `RandomizedLoadBalancing`: balances the load among servers without storing additional information [42]
48. `RndomizedPolynomialFactorization`: finds the roots of a polynomial over a finite field [7]
49. `RandomizedRouting`: finds a route in a boolean hypercube where each node is the destination of exactly one of the 2^n packets being sent [75]
50. `RandomizedTreap`: a full, endogenous random binary treap where each node contains a key and a priority value [68]
51. `RockPaperScissors`: simulates the rock-paper-scissor game
52. `SampleSort`: sorts using divide-and-conquer [22]
53. `SetBalancing`: given $n \times n$-matrix of integers A, finds a n-vector of integers b minimizing $\|Ab\|_\infty$ [57]

54. `SetCover`: finds a minimum set cover: given a 1-0 matrix A, finds a minimized column vector c such that the dot product of each row of A with c is positive while minimizing c [76]
55. `SetIsolation`: finds a sample of the universe U that is disjoint from the subset S but not disjoint from the subset T [40]
56. `SkipList`: implements a skip list [62]
57. `SolovayStrassenPrimalityTest`: determines if a number is prime using Solovay-Strassen's approach [70]
58. `TwoSatisfiability`: given a boolean expression, finds values for the variables such that the expression evaluates to true [59]
59. `UndirectedReachability`: randomly walks through an undirected graph from a source vertex in search for a target vertex [8]
60. `VLSIRouting`: solves a simplified version of the problem of global wiring in gate-arrays [65]

To provide some context, in Table 1 we provide the number of examples of a number of tools. In the table, we count not only discrete time Markov chains, but also continuous time Markov chains, as it is well known that the latter can easily be transformed into the former by abstracting from the timing information.

tool	number	reference	URL
PRISM	36	[48]	www.prismmodelchecker.org
QVBS	23 (1)	[33]	qcomp.org/benchmarks
MRMC	8 (6)	[46]	www.mrmc-tool.org
PARAM	8 (8)	[28]	depend.cs.uni-saarland.de/tools/param
PLASMA	6 (1)	[9]	https://project.inria.fr/plasma-lab
iLTLChecker	5 (5)	[49]	osl.cs.illinois.edu/software/iltl
INFAMY	5 (4)	[29]	depend.cs.uni-saarland.de/tools/infamy
APMC	4 (3)	[34]	github.com/ix-labs/apmc
ePMC	4 (4)	[30]	github.com/ISCAS-PMC/ePMC
IscasMC	4 (4)	[30]	iscasmc.ios.ac.cn
Storm	4 (1)	[15]	www.stormchecker.org
CMurphi	3 (1)	[16]	bitbucket.org/mclab/cmurphi
PVeSta	3 (1)	[4]	maude.cs.uiuc.edu/tools/pvesta
Modest	2 (0)	[32]	www.modestchecker.net

Table 1. All tools are probabilistic model checkers, apart from QVBS which is a benchmark set. The column labelled number contains the number of examples of discrete time and continuous time Markov chains for each tool. The number of examples different from those provided by PRISM is given in parentheses.