PROBABILISTIC MODEL CHECKING OF RANDOMIZED JAVA CODE

SYYEDA ZAINAB FATMI

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
SEPTEMBER 2020

Abstract

Java PathFinder (JPF) and PRISM are the most popular model checkers for Java code and systems that exhibit random behaviour, respectively. JPF, in combination with its extension jpf-probabilistic, extracts the underlying Markov chain of randomized Java code. I developed a new extension of JPF, called jpf-label, which provides users with an easy way to label the states of this Markov chain. Furthermore, I implemented a converter that leads to the first model checking tool that can check probabilistic properties of randomized algorithms implemented in Java, by making it possible to use JPF in conjunction with PRISM.

Probabilistic bisimilarity is a technique used to minimize the state space of a labelled Markov chain in order to combat the state space explosion. I implemented three known algorithms to compute probabilistic bisimilarity for labelled Markov chains. I boosted the performance of these algorithms by improving those areas of the code which are most frequently executed. Moreover, I compared the practical running time and memory consumption of these algorithms with PRISM's.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Franck van Breugel for always being so generous with his time. His patient guidance, advice, and encouragement were invaluable; I have learnt so much from him. It has been an absolute honour to work with him. His unwavering dedication and willingness to help is truly inspiring. I could not have had a better supervisor.

I would like to thank Suprakash Datta for being a member of my supervisory committee and Franz Newland and Richard Wildes for being part of my examination committee. Their stimulating questions, constructive criticism, and detailed feedback were really insightful.

I would like to thank Xiang Chen, Yash Dhamija, and Maeve Wildes for their contributions to this thesis and for a very enjoyable summer. I would also like to thank Qiyi Tang for her contributions and thorough feedback.

Many thanks to Jonathan Ostroff for introducing me to verification (and model checking in particular) with infectious enthusiasm; and also to Hossein Kassiri for

getting me interested in research in the first place.

Last but not least, I am grateful to my family and friends for their constant support and understanding.

Table of Contents

List of Tables

List of Figures

1 Introduction

A *randomized algorithm* is an algorithm that makes random choices during execution, so its behaviour can vary even on a fixed input. Countless current software systems rely on randomized algorithms. It is well-known that randomness is ubiquitous in cryptography to remove predictability [Gen06], thus the security of most modern systems depends on randomized algorithms. Randomness is prevalent in many AI and machine learning algorithms, like stochastic gradient descent [Bot98]. It also has applications in computer games to maintain player interest [SZ04]. These are just a few examples that demonstrate the prominence of randomness in software systems. Furthermore, randomized algorithms can solve problems that cannot be solved by deterministic algorithms, such as the consensus problem [FLP85], and can be drastically more efficient than ordinary deterministic algorithms, for example, in polynomial identity testing [Sch80] where there exists an efficient randomized algorithm, but no deterministic polynomial-time algorithm yet [KI02].

Software testing is most commonly used to show the presence of bugs in software

systems, but it cannot show their absence [Dij69]. Software with randomness may give rise to various executions with potentially different outcomes. Hence, running a test on software with randomness multiple times may not be sufficient, as it is not guaranteed that all possible executions are checked. *Model checking* is a formal verification technique that can be used to show the absence of bugs in the presence of randomness and concurrency, introduced by the Turing award winners Clarke, Emerson [CE81], and Sifakis [QS82].

In the realm of model checking, software is often modelled as a *transition system*. Such a system consists of a nonempty set of *states*, one of which is the *initial state*, and a *transition relation* which specifies those pairs of states that are connected by a transition. A transition system can be viewed as a directed graph with a designated vertex.

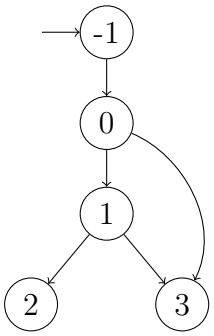


Figure 1.1: A transition system with five states (shown as open circles) and five transitions (shown as arrows).

To capture simple known facts about the states of the modelled software, states

of the transition system are usually *labelled* with a set of atomic propositions. For example, initial and final states may be labelled with the sets $\{initial\}$ and $\{final\}$, respectively. We will discuss other examples of state labellings later. Whenever depicting such a labelled transition system we often use colours to denote the state labelling.

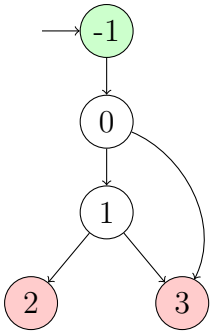


Figure 1.2: A labelled transition system where the initial and final states are labelled, as green and red, respectively.

The *atomic propositions* are often used to express desirable properties of the software system. For example, *initial* and *final* may indicate that a state is initial and final, respectively. These atomic propositions may be used to express properties of the system. Such properties can be formalized in logics, such as linear temporal logic (LTL) [BK08, Chapter 5]. Given a transition system and a property, a *model checker* verifies that the transition system satisfies the property or provides a counterexample, as depicted in Figure 1.3. For example, in LTL one can express the property that from a green state always eventually a red state is reached. This

property is satisfied by the labelled transition system depicted in Figure 1.2.

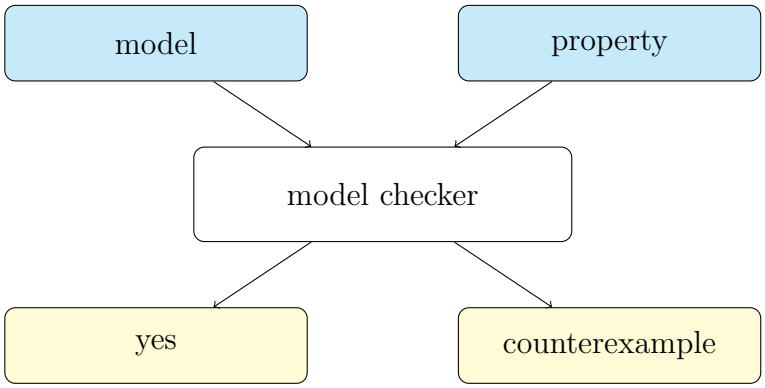


Figure 1.3: Schematic view of the model checking approach. The cyan rectangles are the inputs, while the yellow rectangles are the possible outputs.

The atomic propositions may also be used to minimize the state space by identifying those states of the labelled transition system that are behaviourally equivalent. Behaviour equivalences such as *bisimilarity* [BK08, Chapter 7] rely on the state labelling. The states 2 and 3 of the labelled transition system of Figure 1.2 are bisimilar and therefore can be identified, resulting in the labelled transition system depicted in Figure 1.4.

Java PathFinder (JPF) [VHB$^+$03] is a model checker for Java code. Running JPF on Java code can be viewed as building on the fly a transition system modelling the code. A state of the transition system captures an abstraction of the system state of the Java virtual machine. A transition takes the system from one state to another and represents the execution of a sequence of bytecode instructions.

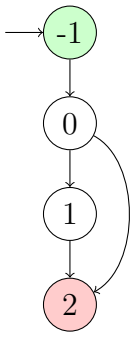


Figure 1.4: The minimized labelled transition system.

JPF can produce a graphical representation of the transition system, as shown in Figure 1.5.

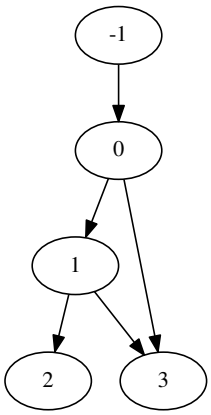


Figure 1.5: A graphical representation produced by JPF.

In Chapter 2, we address the matter of adding labels representing atomic propositions to the transition system, which models the Java code, built by JPF. We accomplish this by developing an extension of JPF, named jpf-label, that allows users to easily label states as specified by custom labelling functions. We can then

provide the resulting labelled transition system, along with a property to check, to a model checker.

The extension of JPF, jpf-probabilistic [ZvB10], assigns probabilities to the transitions of the model, which reflect the random choices in the Java code. By adding probabilities to the transitions, as shown in Figure 1.6, jpf-probabilistic turns the transition system into a (discrete time) Markov chain. Thus, when extended by both jpf-label and jpf-probabilistic, JPF can construct a labelled Markov chain, represented by a transition file and a labelling file.

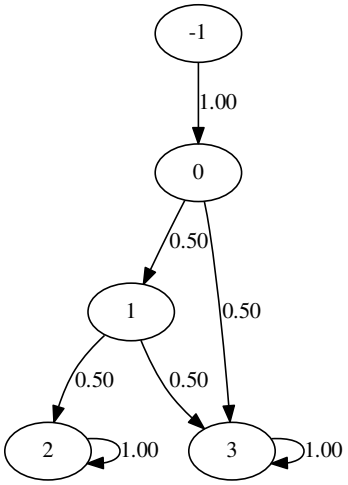


Figure 1.6: A graphical representation produced by JPF, extended with jpf-probabilistic.

PRISM [KNP11] is the most popular probabilistic model checker. As input, it takes a model of a system that exhibits random behaviour and a probabilistic property specified in a logic. The model can be expressed as a labelled Markov

chain. PRISM checks, among other things, whether the model satisfies the property.

We create a converter that transforms the transition and labelling files produced by JPF, with the help of jpf-label and jpf-probabilistic, into a format that can be fed into PRISM together with a probabilistic property. A diagram of this model checking tool can be found in Figure 3.3, which we explore in detail in Chapter 3. Additionally, in Chapter 4, we present several examples to illustrate how the tool can be used. This chapter was developed in collaboration with Xiang Chen, Yash Dhamija, and Maeve Wildes.

JPF, extended with jpf-probabilistic and jpf-label, used in tandem with PRISM, employing the converter, contributes the first tool that can check properties of randomized algorithms implemented in Java. This is significant, because Java is currently one of the most popular programming languages.[1] Since Java bytecode is executed on a Java virtual machine, regardless of the underlying architecture, it is platform-independent, providing programmers with more flexibility. Many of the world's biggest companies use Java.

One of the major challenges of model checking is that the number of states in a model is often too large to check non-trivial properties of the system, as observed by, for example, Clarke [Cla08]. In such a case, the model checker may run out of time or memory before successfully verifying whether the property holds in the

[1] `https://www.tiobe.com/tiobe-index/`

model. This is called the *state space explosion* problem. As mentioned earlier, one method to reduce the state space of a labelled transition system is by identifying those states that are bisimilar.

In Chapter 5, we review the concept of probabilistic bisimilarity, as established in [LS89]. Probabilistic bisimilarity is able to significantly reduce the size of the state space, while preserving the atomic propositions of interest. Thus, we can check properties of the minimized labelled transition system and the results will be valid for the original labelled transition system as well. Moreover, probabilistic bisimilarity can speed up the model checking of Markov chains [KKZJ07].

In Chapter 6, we discuss four different algorithms to compute probabilistic bisimilarity, namely those developed by Buchholz [Buc00], Derisavi, Hermanns and Sanders [DHS03], Valmari and Franceschinis [VF10], and Derisavi [Der07]. The latter has been implemented in the model checker PRISM. We present a few improvements to the algorithms in Chapter 7 and run several experiments to compare the practical performance of these algorithms in Chapter 8.

2 jpf-label: An Extension of Java PathFinder

JPF does not provide an easy way to label the states. In the past, several extensions of JPF, all named jpf-ltl, have been developed that considered state labellings and supported the checking of properties expressed in LTL.[2] Unfortunately, none of these extensions are compatible with the latest version of JPF. In [CC10], Cuong and Cheng describe a tool that given a property expressed in LTL generates an extension of JPF that checks the property. However, an implementation of such a tool is not available [Che19].

Let us briefly discuss how atomic propositions can be defined in the context of Java code. In the literature, the following categories of atomic propositions are distinguished:

- static boolean fields and local boolean variables (jpf-ltl),
- boolean expressions built from static integer fields and local integer variables

[2]Only one version of jpf-ltl is still available. This version is based on the algorithms described in [GL02] and can be found at the URL code.google.com/archive/p/jpfltl/source. Most of the code is more than 15 years old. JPF has changed a lot in the last 15 years, therefore, it should not come as a surprise that this extension is incompatible with the current version of JPF.

(jpf-ltl, [Hol04, Chapter 6], and [KLHN09]),

- method invocations (jpf-ltl and [CC10, KLHN09, SB05]),
- method returns and the values returned ([AGR13, KLHN09]),
- thrown exceptions and the exception types ([KLHN09]), and
- AspectJ pointcuts ([SB05]).

The last one is related to aspect-oriented programming, which is a programming paradigm that is used to modularize crosscutting concerns. AspectJ is a Java extension for aspect-oriented programming.[3] In AspectJ, join points are certain well-defined points in the execution of the program, such as a method call. A pointcut is a set of particular join points, possibly from different classes. Whenever one of the join points described in the pointcut is reached, the code associated with the pointcut is executed. AspectJ pointcuts are very specific to aspect-oriented programming, thus, we do not consider this category, as it has little applicability.

In this chapter, we introduce an extension of JPF called jpf-label. This extension provides an easy way to label states. It implements twelve different ways to label states, including (simplified instances of) all the above mentioned categories apart from AspectJ pointcuts. We discuss these in Section 2.1.1–2.1.6. Our extension

[3]For an introduction to AspectJ, see the AspectJ Programming Guide, which can be found at the URL www.eclipse.org/aspectj/doc/released/progguide.

jpf-label allows users to easily implement their own state labelling as discussed in Section 2.2.

2.1 Using jpf-label

We assume that the reader is already familiar with using JPF.[4] We can use jpf-label to either produce a file that describes the labelling of the states or enhance the graphical representation of the transition system, as already provided by JPF, with colouring the states according to their labelling.

Next, we use the following app to illustrate how jpf-label can label states.

```
1  import java.util.Random;
2
3  public class Field {
4    private static boolean value = true;
5
6    public static void main(String[] args) {
7      Random random = new Random();
8      if (random.nextBoolean()) {
9        Field.value = false;
10       Field.value = true;
11     } else {
12       Field.value = random.nextBoolean();
13     }
14   }
15 }
```

Note that this example is intentionally designed to test the labelling capability of our extension. For instance, in line 9 we set the value of the static field `value` to

[4]Instructions how to use JPF can be found at the URL github.com/javapathfinder/jpf-core/wiki/How-to-use-JPF.

true and then immediately back to false in line 10.

2.1.1 Initial and Final States

To use jpf-label, we create the following application properties file.

```
1 target = Field
2 classpath = <path to the directory containing Field.class>
3 cg.enumerate_random = true
4
5 @using jpf-label
6 listener = label.StateLabelText
7 label.class = label.Initial
```

The system under test (SUT), that is, the Java app to be model checked, is given in line 1. The directory that contains the bytecode of the app needs to be added to JPF's classpath. This is done in line 2. Since the SUT contains randomization and we want JPF to consider both results of `random.nextBoolean()`, we set the property `cg.enumerate_random` to true in line 3. Its default value is false which entails that only one result, namely true, of `random.nextBoolean()` is considered.

Line 5 specifies that we use JPF's extension jpf-label. To specify that JPF should write the labelling of the states to file, we set the `listener` property to the class `label.StateLabelText` in line 6. The property `label.class` captures which states are labelled. In line 7 we specify that only the initial state is labelled.

When we run JPF on this application properties file, a file named `Field.lab` is created with the following content.

```
1 0="init"
2 -1: 0
```

Line 1 specifies that the atomic proposition `init` has index 0. Line 2 captures that state -1 is labelled with 0, that is, state -1 is the initial state.

In general, the format of this file is the one that is used by the model checker PRISM [KNP11]. The first line contains an enumeration of all the labels and their index, which is a non-negative integer. The remaining lines each contain the labelling of a state. This is composed of the state followed by (the indices of) the labels of that state. Note that states are represented by either -1 or a non-negative integer. (PRISM numbers the states starting from zero.) States that do not have any label are not included in the file. The generated file is named `<name of SUT>.lab`.

If we replace line 7 of the above application properties file with

```
7 label.class = label.Initial; label.End
```

then both initial and final states, also known as end states in the JPF context, are labelled, resulting in a file with the following content.

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

By setting the `listener` property to `label.StateLabelDot`, our extension of JPF creates a file named `<name of SUT>.dot` that provides a graphical representation of

the state space: a directed graph the vertices of which are coloured to represent the state labelling. Our extension also creates a file named `<name of SUT>_legend.dot` that explains the mapping between labels and colours. The files are in dot format and can be viewed with the dotty application.[5]

For example, if we replace line 6 and 7 of the above application properties file with

```
6 listener = label.StateLabelDot
7 label.class = label.Initial; label.End
```

and run JPF, two files named `Field.dot` and `Field_legend.dot`, respectively, are created. Opening these files with the dotty application results in the graphical representation depicted in Figure 2.1.

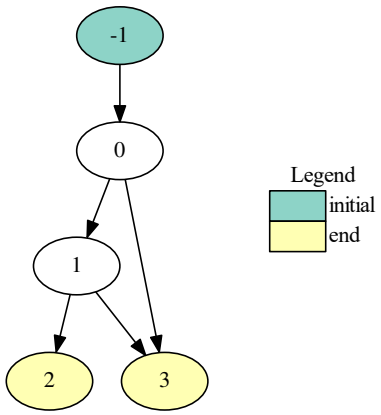


Figure 2.1: Labelling of the initial and final states.

[5]The dotty application can be downloaded from www.graphviz.org.

2.1.2 Boolean Static Fields

Assume that we want to label all states with the value of the boolean static field `value`. Consider the graphical representation presented in Figure 1.5. Let us annotate the app with the states of the corresponding transition system.

```
1  import java.util.Random;
2
3  public class Field {
4    /* state -1 */
5    private static boolean value = true;
6
7    public static void main(String[] args) {
8      Random random = new Random();
9      /* state 0 */
10     if (random.nextBoolean()) {
11       Field.value = false;
12       Field.value = true;
13     } else {
14       /* state 1 */
15       Field.value = random.nextBoolean();
16     }
17     /* state 2 if Field.value is false,
18        state 3 otherwise */
19   }
20 }
```

If we label each state with the value of the boolean static field `value`, then we obtain the graphical representation presented in Figure 2.2. In the initial state -1, JPF has not yet initialized the field `value`. Hence, `value` has the default value, which is false. Therefore, this state should be labelled with false. The transition from state -1 to state 0 corresponds to the sequence of bytecode instructions that includes the initialization of the field `value`, the invocation of the `main` method,

and the execution of this method up to `random.nextBoolean()` in line 10. Hence, in state 0 `value` has the value true and, hence, this state should be labelled with true.

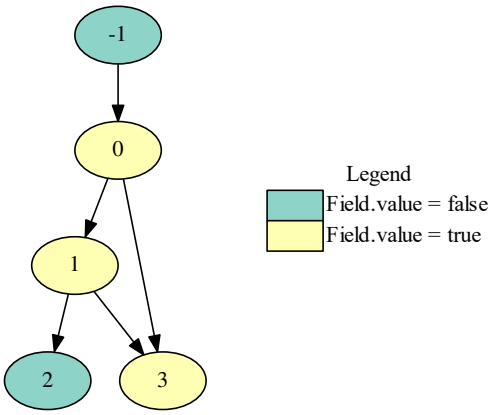


Figure 2.2: Labelling the states of Figure 1.5 with the value of the boolean static field `value`.

The two transitions that leave state 0 correspond to the two possible results of `random.nextBoolean()` on line 10. State 1 is reached if the result is false, that is, if line 15 is reached. In this state, the field `value` is true and, therefore, also this state should be labelled with true. The two outgoing transitions of state 1 represent the possible results of `random.nextBoolean()` on line 16. If the result is false then the final state 2 is reached. The transition between state 1 and state 2 includes the assignment of false to `value` and, hence, the state 2 should be labelled with false. If the result is true then the final state 3 is reached. Since the field `value` is true in state 3, this state should be labelled with true.

Finally, we consider the other transition from state 0 that corresponds to the execution of line 11–12. This results in a final state in which the field `value` is true, that is, state 3.

If we inspect the app, it is obvious that there exists an execution in which the value of the field `value` changes from true to false and subsequently back to true. However, this information is lost in the labelled transition system of Figure 2.2. That is, the labelled transition system does not have a path segment with one or more states labelled with true followed by one or more states labelled with false followed by one or more states labelled with true. The labelled transition system is lacking this information because the transition from state 0 to state 3, which corresponds to he execution of line 11-12, contains two assignments to the field `value`, changing the field `value` from true to false and then back to true. To address this, we break the transition after the first assignment to `value` on line 11 and introduce an intermediate state. In this new state, the field `value` is false and, hence, this state should be labelled with false.

More generally, we break a transition whenever the value of the field `value` is changed. The initialization of `value` on line 5 corresponds to a `PUTSTATIC` bytecode instruction. Since the value of the field has changed, we break this transition from state -1 to state 0 into two transitions immediately after that `PUTSTATIC` and introduce an intermediate state (see Figure 2.3). Hence, in the new state and in

state 0, the field `value` is true and, therefore, these states should be labelled with true.

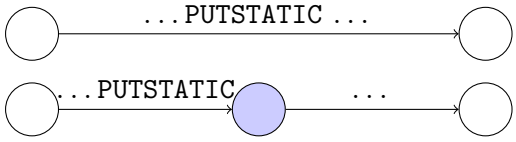


Figure 2.3: Transition is broken into two to observe the effect of `PUTSTATIC`.

The transition between state 1 and state 2 in Figure 1.5 includes the assignment of false to `value`, which also corresponds to a `PUTSTATIC` bytecode instruction. Since the value of the field has changed, we break the transition into two after the `PUTSTATIC`, creating an intermediate state, which should be labelled with false as the field `value` is false.

We break the transition from state 0 to state 3 in Figure 1.5 again after the second assignment to `value` on line 12, creating another intermediate state. In this new state, the field `value` is true, thus, this state should be labelled with true.

If the result of `random.nextBoolean()` on line 16 is true, then the final state 3 in Figure 1.5 is reached. This time we do not break the transition after the assignment to the field, as `value` remained true.

Note that some of the transitions in Figure 1.5 need not be broken as the intermediate states and the target state are labelled in the same way. For example, the transition from state -1 to state 0 in Figure 1.5 need not be broken since the inter-

mediate state, state 0 in Figure 2.4, and the target state, state 1 in Figure 2.4, are both labelled with true. Avoiding these transitions to be broken would complicate the code of jpf-label significantly. We leave this as a topic for further research.

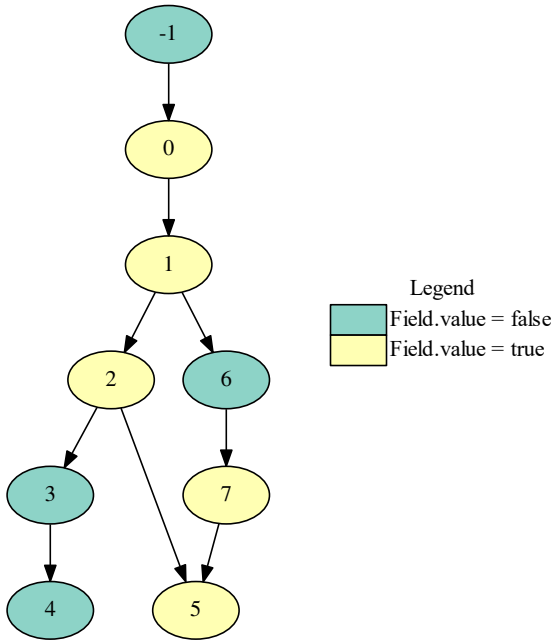


Figure 2.4: Labelling states with the value of the field `value`.

To obtain this graphical representation, we run JPF with the following application properties file.

```
target = Field
classpath = <path to the directory containing Field.class>
cg.enumerate_random = true

@using jpf-label
listener = label.StateLabelDot
label.class = label.BooleanStaticField
label.BooleanStaticField.field = Field.value
```

The class `label.BooleanStaticField` labels all states with the values of the boolean static fields specified by the property `label.BooleanStaticField.field`. In this example the boolean static field `Field.value` is considered. In general, the fully qualified name of the class (see [GJS+15, Section 6.7]) and its field are specified. Running JPF on the above application properties file produces the graphical representation depicted in Figure 2.4. If we compare Figure 2.4 with Figure 1.5, we notice four extra states, namely states 0, 3, 6 and 7 in Figure 2.4. These extra states correspond to the above mentioned breaks of the transitions.

To provide an example in which multiple fields are labelled, we will use the following app.

```
1  import java.util.Random;
2
3  public class MultipleFields {
4    private static boolean one = true;
5    private static boolean two = true;
6    private static boolean three = false;
7
8    public static void main(String[] args) {
9      Random random = new Random();
10     if (random.nextBoolean()) {
11       MultipleFields.three = false;
12       MultipleFields.two = true;
13     } else {
14       MultipleFields.three = true;
15       MultipleFields.two = false;
16     }
17   }
18 }
```

If we run JPF without any labelling, we obtain the graphical representation

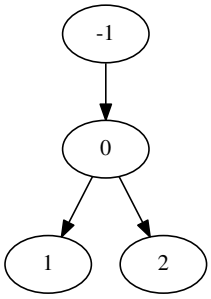


Figure 2.5: The state space of the class `MultipleFields`.

shown in Figure 2.5. Suppose we would like to consider all three boolean static fields, that is `one`, `two`, and `three`. In the initial state -1, none of the boolean fields have been initialized, thus they have the default value of false and the state should be labelled with three labels indicating that all three fields are false. The transition from state -1 to state 0 corresponds to the sequence of bytecode instructions including the initialization of the three boolean fields, the invocation of the `main` method and its execution up to line 10. After the initialization of `one` to true in line 4, we break the transition in the manner explained in the previous example, creating a new state. This new state has a label indicating that the field `one` is true and two labels indicating that the fields `two` and `three` are false. We break the transition again after the initialization of `two` to true in line 5, creating another new state. This state has two labels indicating that both `one` and `two` are true and a label indicating that `three` is false. We do not break the transition after the initialization of the field `three` to false, since its value does not change. There is

no further modification of the fields until line 10, thus state 0 has the same three labels as the preceding state.

The two outgoing transitions from state 0 represent the two possible results of `random.nextBoolean()` in line 10. If the result is false, we reach state 1. The transition from state 0 to state 1 includes the bytecode instructions that correspond to lines 14–18. The assignment to the field `three` in line 14 changes the value of the field from false to true. Hence, we break the transition just after the assignment, introducing an intermediate state. This new state has three labels, signifying that all of the fields `one`, `two`, and `three` are true. The assignment to `two` on line 15 changes the value of the field from true to false. Thus we break the transition just after this assignment, introducing a second intermediate state. This new state has two labels, indicating that the fields `one` and `three` are true and one label indicating that the field `two` is false. There is no further modification of the fields until line 18, thus state 1 has the same three labels as the preceding state.

However, if the result of `random.nextBoolean()` in line 10 is true, then lines 11–12 are executed and state 2 is reached. We do not break the transition after the assignment to `three` in line 11 or the assignment to `two` in line 12, because these assignments do not change the value of the fields. Hence, state 2 will have the same three labels as state 0, indicating that the boolean static fields `one` and `two` are true and the boolean static field `three` is false.

In order to obtain a graphical representation of the labelling of the boolean static fields `one`, `two`, and `three`, we use the following application properties file.

```
1 target = MultipleFields
2 classpath = <path to the directory containing
     ↪ MultipleFields.class>
3 cg.enumerate_random = true
4
5 @using jpf-label
6 listener = label.StateLabelDot
7 label.class = label.BooleanStaticField
8 label.BooleanStaticField.field = MultipleFields.one;
     ↪ MultipleFields.two; MultipleFields.three
```

Running JPF with these application properties gives rise to the graphical representation of the labelled state space illustrated in Figure 2.6. Additionally, we can obtain the textual representation of the labelled state space by replacing line 6 of the application properties file with

```
6 listener = label.StateLabelText
```

which generates the file `MultipleFields.lab` with the following content.

```
1 0="false__MultipleFields_one" 1="false__MultipleFields_two"
     ↪ 2="false__MultipleFields_three"
     ↪ 3="true__MultipleFields_one"
     ↪ 4="true__MultipleFields_two"
     ↪ 5="true__MultipleFields_three"
2 -1: 0 1 2
3 0: 1 2 3
4 1: 2 3 4
5 2: 2 3 4
6 3: 3 4 5
7 4: 1 3 5
8 5: 1 3 5
9 6: 2 3 4
```

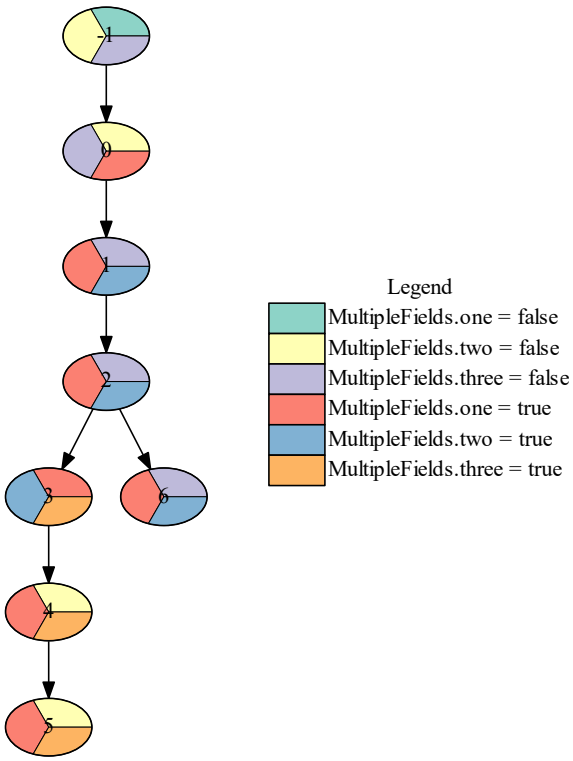


Figure 2.6: Labelling states with the values of the fields `one`, `two`, or `three`.

The textual representation of the labelling shown above describes which of the boolean static fields are true and which are false in each state.

Similarly, we have implemented the class `IntegerStaticField` that can be used to label all states with the values of particular integer static fields. These fields are specified by the property `label.IntegerStaticField.field`.

2.1.3 Integer Local Variables

The class `IntegerLocalVariable` can be used to label all states, within the scope of a specified integer local variable, with the integer value of that local variable.[6] For example, consider the following app.

```
1 public class Variable {
2   public static void main(String[] args) {
3     int value = 0;
4     value++;
5     value = value % 5;
6     value -= 2;
7   }
8 }
```

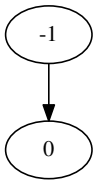


Figure 2.7: The state space of the class `Variable`.

If we run JPF without our extension jpf-label, then we can generate the graphical representation of the state space depicted in Figure 2.7. State -1 is the initial state and state 0 is the final state. The transition from state -1 to state 0 corresponds to the sequence of all the bytecode instructions of the `Variable` app.

Assume that we want to keep track of the value of the local variable `value`. Since this local variable is only declared in line 3, in the initial state -1 the local variable

[6] Compile the Java code with the -g option.

`value` is not yet within scope and, hence, this state is not labelled. The transition from state -1 to state 0 contains several bytecode instructions that manipulate `value`. Therefore, we break the transition. Note that we try to break the transition only if the value of the local variable changes, in order to minimize the size of the state space. When we reach line 3, the local variable `value` is declared and is assigned the value zero. Hence, we break the transition and the new state is labelled with zero. In line 4, `value` is incremented by one. Therefore, we break the transition again and label the new state with one. In line 5, `value` is assigned to one modulus five, making it one. Since `value` remains the same value, we do not break the transition. In line 6, `value` is decremented by two. Hence, once again we break the transition. The new state is labelled with negative one. When we reach the final state, the `main` method has been exited and, hence, the local variable `value` is not within scope any more. Therefore, the final state is not labelled.

To use our extension jpf-label to label those states within the scope of the local variable `value`, with the integer value of `value`, we introduce the following application properties file.

```
1 target = Variable
2 classpath = <path to the directory containing Variable.class>
3
4 @using jpf-label
5 listener = label.StateLabelDot
6 label.class = label.IntegerLocalVariable
7 label.IntegerLocalVariable.variable =
    ↪ Variable.main(java.lang.String[]):value
```

The variable is specified by setting the property `label.IntegerLocalVariable.variable`, as shown in line 7, using the fully qualified name of the class, the method signature, and the variable name. Note that the argument types in the method signature are not required unless the method is overloaded. In this example, we examine the local variable `value` of the method `main(java.lang.String[])` of the class `Variable`.

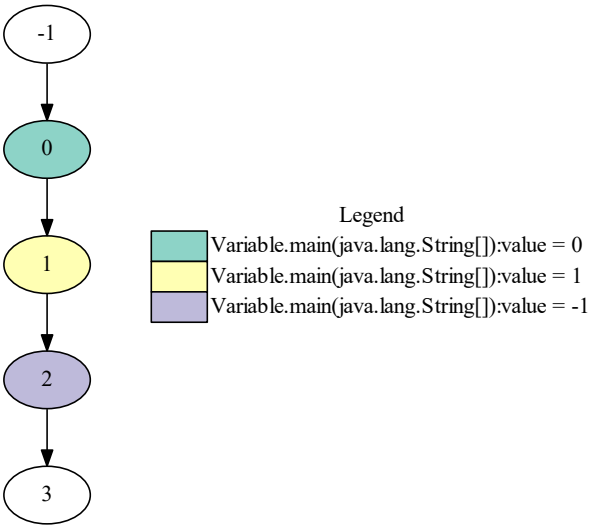


Figure 2.8: Labelling of states with the value of the local variable `value`.

If we run JPF on the above application properties file then we obtain the labelled transition system depicted in Figure 2.8. If we compare Figure 2.8 with Figure 2.7 we notice three extra states, namely states 0, 1 and 2 in Figure 2.8. These extra states correspond to the above mentioned breaks of transitions.

When the `listener` property is set to `label.StateLabelText`, a file named

`Variable.lab` is created with the following contents, representing the labelling of states 0, 1, and 2.

```
0="0__Variable_main___3Ljava_lang_String_2__V__value"
   ↪ 1="1__Variable_main___3Ljava_lang_String_2__V__value"
   ↪ 2="minus1__Variable_main___3Ljava_lang_String_2__V__value"
0: 0
1: 1
2: 2
```

As shown above, the textual representation of the labelling uses label names similar to the name mangling used in the Java native interface [Lia99]. The label with ID zero captures that the local variable `value` of the method `main`, which takes an argument of type `String[]` and has a return type of `void`, of the class `Variable` has a value of zero. Note that if the value of the local variable is negative, the label name is prefixed by the string `minus`.

Similarly, we have implemented the class `BooleanLocalVariable` that can be used to label those states within the scope of a boolean local variable. The states are labelled with the value of the boolean local variable, namely `true` or `false`, and its mangled signature.

2.1.4 Method Invocations and Returns

The class `InvokedMethod` allows us to label those states in which a specified method is invoked, while the classes `ReturnedBooleanMethod`, `ReturnedIntegerMethod`, and `ReturnedVoidMethod` allow us to label those states in which a specified boolean,

integer, or void method returns, respectively. For example, consider the following app.

```
1  import java.util.Random;
2
3  public class Method {
4    private static int value;
5
6    public static void setValue(int value) {
7      Method.value = value;
8    }
9
10   public static void main(String[] args) {
11     Random random = new Random();
12     if (random.nextBoolean()) {
13       Method.setValue(2);
14     }
15   }
16 }
```

We create the following application properties file.

```
1 target = Method
2 classpath = <path to the directory containing Method.class>
3 cg.enumerate_random = true
4
5 @using jpf-label
6 listener = label.StateLabelDot
7 label.class = label.InvokedMethod
8 label.InvokedMethod.method = Method.setValue(int)
```

The states in which the static method `Method.setValue(int)`, which is specified by the property `label.InvokedMethod.method`, is invoked will be labelled. When we run JPF on this application properties file, we obtain the graphical representation shown in Figure 2.9.

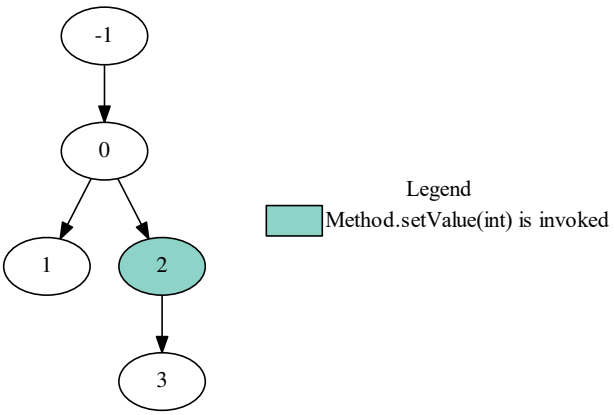


Figure 2.9: Labelling of states in which the method `setValue` is invoked.

The transition from the initial state -1 to state 0 corresponds to the series of bytecode instructions that includes the initialization of the integer static field `value`, the invocation of the `main` method, and the execution of this method up to `random.nextBoolean()` in line 12. Evidently, the method `setValue` is not invoked in state -1 and state 0, thus these states are not labelled.

The two transitions that leave state 0 correspond to the two possible results of `random.nextBoolean()` on line 12. If the result is false, only line 15 and 16 are executed and state 1 is reached. Since the static method `setValue` is not invoked in those lines, state 1 is not labelled.

If the result of `random.nextBoolean()` on line 12 is true, the next line to be executed is line 13. Since the static method `setValue` will be invoked in line 13, we break the transition just before the invocation and introduce a new state, namely state 2. The method is invoked in this state and, therefore, state 2 is labelled.

Finally, the transition between state 2 and state 3 encompasses the invocation, execution, and return of the `setValue` method, and the return of the `main` method also. Since we arrive at state 3 after the execution of line 13–16, the state is not labelled.

If we want to label the return of the void method `setValue` instead, we can replace line 7 and 8 with

```
7 label.class = label.ReturnedVoidMethod
8 label.ReturnedVoidMethod.method = Method.setValue(int)
```

resulting in the labelled transition system depicted in Figure 2.10.

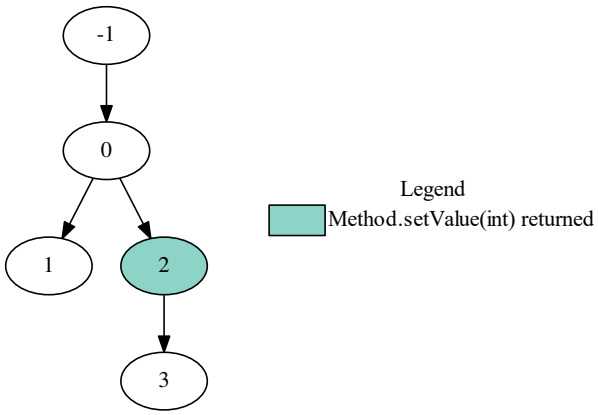


Figure 2.10: Labelling of states in which the method `setValue` has returned.

The states -1, 0, and 1 and the transitions between them remain as described above. However, if the result of `random.nextBoolean()` on line 12 is true, we consider the transition from state 0 to state 2. This transition includes the invocation, execution and return of the void method `setValue`. After the return we break

the transition and introduce state 2. The method has returned in this state, thus state 2 is labelled. Finally, the transition from state 2 to state 3 refers to the sequence of bytecode instructions that includes the return of the `main` method, that is, lines 14–16. Hence, state 3 is not labelled.

In order to label the states in which the method is invoked as well as those states in which the method has returned, we can use the following application properties file.

```
1 target = Method
2 classpath = <path to the directory containing Method.class>
3 cg.enumerate_random = true
4
5 @using jpf-label
6 listener = label.StateLabelDot
7 label.class = label.InvokedMethod; label.ReturnedVoidMethod
8 label.InvokedMethod.method = Method.setValue(int)
9 label.ReturnedVoidMethod.method = Method.setValue(int)
```

Running JPF with these application properties produces the labelled state space illustrated in Figure 2.11. To obtain the text representation, we replace line 6 with

```
6 listener = label.StateLabelText
```

When JPF is run with these application properties, a file named `Method.lab` is created with the following content.

```
1 0="invoked__Method_setValue__I__V"
     ↪ 1="returned__Method_setValue__I__V"
2 2: 0
3 3: 1
```

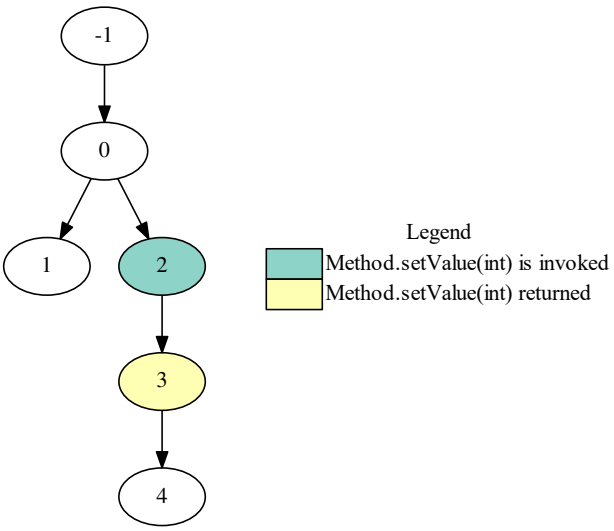


Figure 2.11: Labelling of states in which the method `setValue` is invoked and also when it has returned.

The transition from state -1 to state 0 and the transition from state 0 to state 1 are unaffected and thus these three states are not labelled, as explained previously. The transition from state 0 to state 2 occurs when the result of `random.nextBoolean()` on line 12 is true. In this state, the method `setValue` of line 13 is invoked. The class `InvokedMethod` breaks the transition just before that invocation, creating state 2. The method is invoked in state 2, therefore, the state is labelled. The transition from state 2 to state 3 encompasses the invocation of the method `setValue`, the execution of this method, and its return. The latter causes a break of the transition, creating state 3. The void method has returned in state 3, therefore, the state is labelled. Lastly, the transition from state 3 to state 4 refers to the sequence of bytecode instructions that includes the return of the `main` method,

that is, lines 14–16. Hence, state 4 is not labelled.

To provide an example in which the return of a non-static boolean method is labelled, we will use the following app.

```
1  import java.util.Random;
2
3  public class BooleanMethod {
4    public boolean getRandom() {
5      Random random = new Random();
6      return random.nextBoolean();
7    }
8
9    public static void main(String[] args) {
10     BooleanMethod instance = new BooleanMethod();
11     instance.getRandom();
12   }
13 }
```

We create the following application properties file to label those states in which the method `getRandom()` has returned, with its return value.

```
1  target = BooleanMethod
2  classpath = <path to the directory containing
     ↪ BooleanMethod.class>
3  cg.enumerate_random = true
4
5  @using jpf-label
6  listener = label.StateLabelDot
7  label.class = label.ReturnedBooleanMethod
8  label.ReturnedBooleanMethod.method =
     ↪ BooleanMethod.getRandom()
```

Running JPF with these application properties creates the graphical representation of the labelling shown in Figure 2.12.

The transition from the initial state -1 to state 0 corresponds to the series of

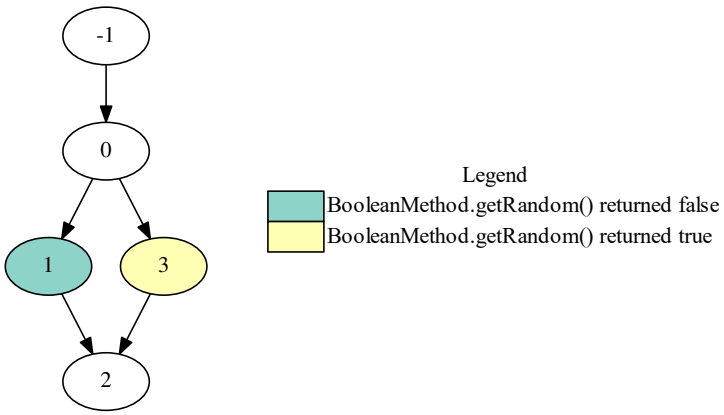


Figure 2.12: Labelling of states with the returned value of the method `getRandom`.

bytecode instructions that includes the invocation of the `main` method, the initialization of the `BooleanMethod` object `instance`, the invocation of the `getRandom` method, and the execution of this method up to `random.nextBoolean()` in line 7. Since the method `getRandom` has not yet returned in state -1 and state 0, these states are not labelled.

The two transitions that leave state 0 correspond to the two possible results of `random.nextBoolean()` on line 7. If the result is false, the method `getRandom` returns false, the transition is broken and state 1 is created. Since the boolean method `getRandom` has returned false, state 1 is labelled. The transition between state 1 and state 2 comprises the return of the `main` method, thus state 2 is not labelled.

If the result of `random.nextBoolean()` on line 7 is true, the method `getRandom` returns true and the transition is broken, introducing state 3. Since the boolean

method `getRandom` has returned true, state 3 is labelled, but with a different label to that of state 1 as the values returned are distinct. Finally, the transition from state 3 to state 2 represents the return of the `main` method, hence state 2 is not labelled.

To get the textual representation of the labelling described above, we modify line 6 of the application properties file as follows.

```
6 listener = label.StateLabelText
```

Running JPF on the modified application properties results in the generation of a file named `BooleanMethod.lab` with the following content.

```
1 0="false__BooleanMethod_getRandom____Z"
    ↪ 1="true__BooleanMethod_getRandom____Z"
2 1: 0
3 3: 1
```

Note that there are two separate labels for those states in which the method `getRandom` returns `false` and for those states in which the method `getRandom` returns `true`.

Similarly, the class `ReturnedIntegerMethod` labels those states in which a specified integer method returns. The labelling format is similar to that described above, that is, the label consists of the returned value followed by the mangled method signature. Note that if the returned value is negative, the label is prefixed by the string `minus`.

2.1.5 Exceptions and Errors

The class `ThrownException` enables us to label the states in which an error or exception has been thrown. For example consider the following app.

```
import java.lang.annotation.AnnotationFormatError;
import java.util.Random;
import java.util.zip.DataFormatException;

public class Throw {
  public static void main(String[] args) {
    Random random = new Random();
    try {
      if (random.nextBoolean()) {
        throw new DataFormatException("exception");
      } else {
        throw new AnnotationFormatError("error");
      }
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }
}
```

To obtain a graphical representation, we create the following application properties file.

```
target = Throw
classpath = <path to the directory containing Throw.class>
cg.enumerate_random = true

@using jpf-label
listener = label.StateLabelDot
label.class = label.ThrownException
label.ThrownException.exception =
   ↪ java.lang.annotation.AnnotationFormatError;
   ↪ java.util.zip.DataFormatException
```

The errors and exceptions which will be labelled should be specified by the property `label.ThrownException.exception`, using the fully qualified names of the exception and error classes. In this example, we consider the error `java.lang.annotation.AnnotationFormatError` and the exception `java.util.zip.DataFormatException`. When JPF is run with the above application properties file, we get the graph displayed in Figure 2.13.

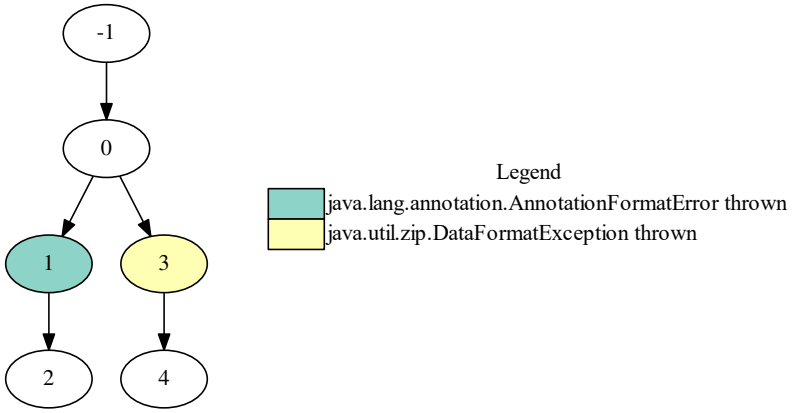


Figure 2.13: Labelling of states in which the specified error or exception has been thrown.

To obtain the textual representation of the labelled state space, we set the property `listener` in line 6 to `label.StateLabelText` instead. Then when JPF is run with the application properties file, we get a text file, named `Throw.lab`, with the following information.

```
1 0="thrown__java_lang_annotation_AnnotationFormatError"
    ↪ 1="thrown__java_util_zip_DataFormatException"
2 1: 0
3 3: 1
```

The transition from the initial state -1 to state 0 corresponds to the sequence of bytecode instructions which includes the invocation of the `main` method and its execution up to `random.nextBoolean()` in line 9. The two transitions leaving state 0 represent the possible outcomes of `random.nextBoolean()` on line 9. If the result is false, an `AnnotationFormatError` is thrown in line 12. The transition is broken after the exception has been thrown, creating state 1. Since a specified error has been thrown in state 1, it is labelled. The transition from state 1 to state 2 corresponds to the series of bytecode instructions including the `catch` block and the return of the `main` method. Similarly, if the result of `random.nextBoolean()` on line 9 is true, a `DataFormatException` is thrown in line 10, after which the transition is broken, creating state 3. Since a specified exception has been thrown in state 3, it is labelled. Finally, the transition from state 3 to state 4 corresponds to the series of bytecode instructions including the `catch` block and the return of the `main` method.

Our `ThrownException` class only labels states in which an exception has been explicitly thrown, that is, using Java's `throw` statement. For example, the expression `1 / 0` implicitly throws an `ArithmeticException`, which is not detected by our class.

2.1.6 Synchronized Methods

Stolz and Bodden [SB05] mention synchronization points as a potential source for labelling states. The class `SynchronizedStaticMethod` allows us to label the states

in which a thread acquires or releases the lock on a synchronized static method. For example, consider the following app.

```
1  public class Synchronized {
2    private static double value;
3
4    public synchronized static void setValue(double value) {
5      Synchronized.value = value;
6    }
7
8    public static void main(String[] args) {
9      Synchronized.setValue(0.0);
10     Synchronized.setValue(Math.PI);
11   }
12 }
```

We create the following application properties file, in order to label the acquiring and releasing of the lock on the static synchronized method `setValue`.

```
1  target = Synchronized
2  classpath = <path to the directory containing
       ↪ Synchronized.class>
3
4  @using jpf-label
5  listener = label.StateLabelDot
6  label.class = label.SynchronizedStaticMethod
7  label.SynchronizedStaticMethod.method =
       ↪ Synchronized.setValue(double)
```

When JPF is run on the above application properties file, the labelled transition system depicted in Figure 2.14 is generated.

The initial state -1 is not labelled as the lock has not been obtained yet. The transition from state -1 to state 0 represents the series of bytecode instructions that include the initialization of the field `value`, the invocation of the `main` method,

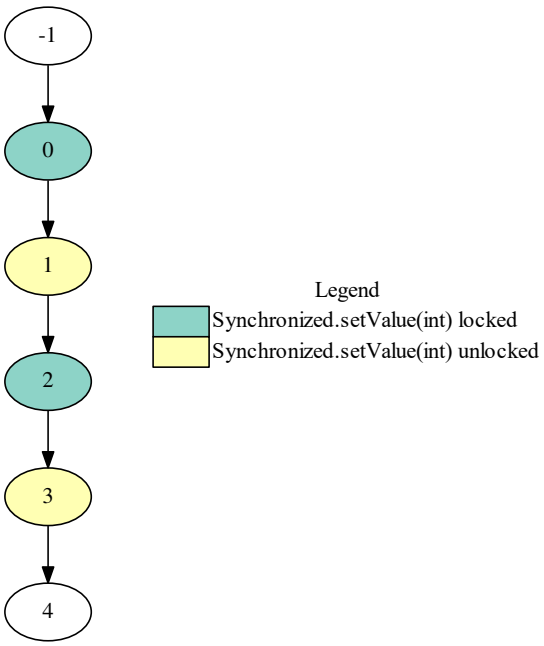


Figure 2.14: Labelling of states in which the synchronized method `setValue` acquires or releases the lock.

and the execution of this method up to and including line 8. In the next line to be executed, line 9, the synchronized static method `setValue` will be invoked. Since the method `setValue` has a `synchronized` modifier, the lock must be obtained before the method is invoked. Thus, we break the transition just before the invoke, creating state 0, which is labelled as the acquire of the lock.

The transition between state 0 and state 1 represents the series of bytecode instructions that include the invocation, execution, and return of the synchronized static method `setValue`. Since the method has a `synchronized` modifier, the lock must be released after the method has returned. Therefore, we break the transition

after the return, creating state 1, which is labelled as the release of the lock.

Similarly, we break the transition just before the second invocation of the synchronized static method `setValue` in line 10, generating state 2, which is labelled as the acquire of the lock. The transition between state 2 and state 3 encompasses the invocation, execution, and return of this method. Once again, we break the transition after the return of the `setValue` method, generating state 3, which is labelled as the release of the lock. The final transition from state 3 to state 4 corresponds to the return of the `main` method, thus state 4 is not labelled.

Modifying line 6 the application properties file to

```
6 listener = label.StateLabelText
```

results in the generation of the file `Synchronized.lab` with the following text.

```
1 0="locked__Synchronized_setValue__I__V"
    ↪ 1="unlocked__Synchronized_setValue__I__V"
2 0: 0
3 1: 1
4 2: 0
5 3: 1
```

The textual representation of the labelling clearly shows the sequence obtaining and releasing of the lock.

2.1.7 Summary

In summary, we have developed the following two listeners.

- `StateLabelText`: writes the states labelling to a file.
- `StateLabelDot`: writes a graphical representation of the labelled transition system to a file.

Both listeners support the following classes to label states. For some classes an additional property needs to be specified as indicated below.

- `Initial`: labels the initial state.
- `End`: labels the final states (also known as end states in JPF).
- `BooleanStaticField`: labels states with the value of the boolean static field specified by the property `label.BooleanStaticField.field`.
- `IntegerStaticVariable`: labels states with the value of the integer static field specified by the property `label.IntegerLocalVariable.variable`.
- `BooleanLocalVariable`: labels those states that are within the scope of the boolean local variable specified by the property `label.BooleanLocalVariable.variable`, with the value of that variable.
- `IntegerLocalVariable`: labels those states that are within the scope of the integer local variable specified by the property `label.IntegerLocalVariable.variable`, with the value of that variable.

- `InvokedMethod`: labels those states in which the method specified by the property `label.InvokedMethod.method` is invoked.

- `ReturnedVoidMethod`: labels those states in which the void method specified by the property `label.ReturnedVoidMethod.method` has returned.

- `ReturnedBooleanMethod`: labels those states in which the boolean method specified by the property `label.ReturnedBooleanMethod.method` has returned, with the return value.

- `ReturnedIntegerMethod`: labels those states in which the integer method specified by the property `label.ReturnedIntegerMethod.method` has returned, with the return value.

- `ThrownException`: labels those states in which the exception or error of the type specified by the property `label.ThrownException.type` has been thrown.

- `SynchronizedStaticMethod`: labels those states in which the synchronized static method specified by the property `label.SynchronizedStaticMethod.method` acquires and has released the lock.

All the above mentioned classes are part of the package `label`. The standard Java APIs of jpf-label can be generated by running gradle with the argument api (see Appendix A).

In Table 2.1, we identify the relevant bytecode instructions for those labelling classes above that rely on breaking the transition in order to observe the required events. We also specify whether the transition is broken before or after the mentioned bytecode instructions. In Section 2.3 we go into more detail.

Labelling class	Break before	Break after
`BooleanStaticField`		`PUTSTATIC`
`IntegerStaticVariable`		`PUTSTATIC`
`BooleanLocalVariable`		`ISTORE`
`IntegerLocalVariable`		`ISTORE` or `IINC`
`InvokedMethod`	`INVOKESTATIC` or `InstanceInvocation`	
`ReturnedVoidMethod`		`RETURN`
`ReturnedBooleanMethod`		`IRETURN`
`ReturnedIntegerMethod`		`IRETURN`
`ThrownException`		`ATHROW`
`SynchronizedStaticMethod`	`INVOKESTATIC`	`ReturnInstruction`

Table 2.1: The bytecode instructions before or after which we break the transition for each labelling class.

2.2 Extending jpf-label

The extension `jpf-label` allows users to easily define their own labelling functions to label states with desired atomic propositions, and to readily construct their own format for the output of the labelling. The labelling functions can be divided into two categories, that is, those which depend on transitions and those which do not. We will discuss the latter first.

2.2.1 Labelling States

In order to label states, we extend the abstract class `StateLabelMaker` in the package `label` and implement the following methods.

1. The static method `getInstance` returns an instance of the class. This method takes as an argument JPF's `Config` object, from which properties such as those specified in an application properties file may be accessed.

2. The method `getStateLabels` is executed whenever a new state is reached by JPF. This method takes as an argument JPF's `Search` object, which provides access to the internals of JPF, and returns a set of labels for the new state. A label consists of a name and a description and is represented by the class `Label`.

For example, assume that we want to label final states with the label `"end"`. We

create the following class `End` which extends `StateLabelMaker`.

```
1  package label;
2
3  import gov.nasa.jpf.Config;
4  import gov.nasa.jpf.search.Search;
5
6  import java.util.HashSet;
7  import java.util.Set;
8
9  /**
10  * A labelling function for final states.
11  */
12 public class End extends StateLabelMaker {
13
14   /**
15    * Initializes this labelling function.
16    */
17   private End() {}
18
19   /**
20    * Creates an End object.
21    *
22    * @param configuration JPF's configuration.
23    * @return an instance of this class.
24    */
25   public static End getInstance(Config configuration) {
26     return new End();
27   }
28
29   /**
30    * Whenever the search advances to the next state, returns
31    * the labels associated with the new state.
32    *
33    * @param search JPF's search.
34    * @return the set of labels for the current state.
35    */
36   @Override
37   public Set<Label> getStateLabels(Search search) {
38     Set<Label> labels = new HashSet<Label>();
39     if (search.isEndState()) {
40       labels.add(new Label("end", "end"));
```

```
41      }
42      return labels;
43    }
44  }
```

In addition to the default constructor, the class contains the methods `getInstance` and `getStateLabels` mentioned above. The static method `getInstance` on lines 25–27 returns an instance of the class `End`. The method `getStateLabels` returns a set of labels. To determine whether the current state is a final state, we consult JPF. The method call `search.isEndState()` returns true if and only if the current state is a final state. Hence, if the current state is a final state, `search.isEndState()` in line 39 returns true and a new `Label`, with name and description `end`, is added to the set of labels for the state in line 40. Otherwise, `search.isEndState()` in line 39 returns false and an empty set is returned.

The class `Initial` labels the initial state with the label `"init"`. This class also extends the abstract class `StateLabelMaker` and is implemented similarly (see Figure 2.15).

2.2.2 Labelling Transitions

Transitions between states represent the execution of a sequence of bytecode instructions. In the event that we wish to observe a specific instruction, we may break the transition either before (see Figure 2.16) or after (see Figure 2.17) the

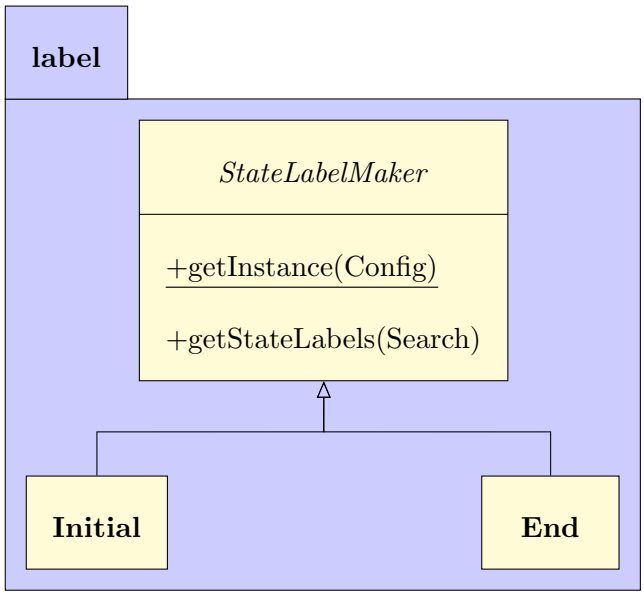


Figure 2.15: UML diagram of the state labelling classes.

point of interest and label the newly generated state.

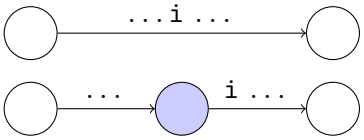


Figure 2.16: Transition is broken into two before bytecode instruction `i` and the new state is labelled.

We can define state labellings in this way by extending the abstract class `TransitionLabelMaker`, which extends `StateLabelMaker`. It is necessary for the subclass to contain the `getInstance` method, which is described in Section 2.2.1. The following methods could be optionally overridden.

1. The method `getStateLabels` is also already described in Section 2.2.1.

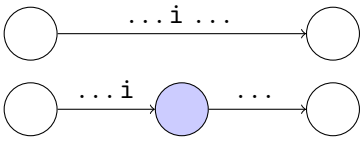


Figure 2.17: Transition is broken into two after bytecode instruction `i` and the new state is labelled.

2. The method `breakAfter` is executed just after a bytecode instruction has been executed by JPF. This method takes as an argument that instruction. If the transition must be broken after this instruction, the method should return the set of labels for the resulting state. Otherwise, the method should return `null`, which is the default.

3. The method `breakBefore` is executed just before a bytecode instruction is executed by JPF. This method takes as an argument that instruction. If the transition must be broken before this instruction, the method should return the set of labels for the resulting state. Otherwise, the method should return `null`, which is the default.

4. The method `beforeInstruction` is executed just before a bytecode instruction is about to be executed by JPF, which allows the user to access information about the state before the instruction is executed. This method takes as an argument the instruction to be executed.

Note that when the transition is broken by either of the methods `breakAfter` or

`breakBefore`, the newly created state will be labelled with the set of labels returned by the method which caused the break in the transition as well as the set of labels returned by `getStateLabels`. Furthermore, after an instruction is executed, if both `breakAfter` and `breakBefore` indicate that the transition must be broken at this point, only a single new state will be created, labelled with the sets of labels returned by both of these methods as well as the set of labels returned by `getStateLabels`.

Our extension already supports a range of atomic propositions that rely on breaking the transitions at certain points of interest. For example, it allows us to identify those states in which a specific boolean static field is true, those states in which a specific method is invoked or returns, those states in which a specific error is thrown, etc. These labelling classes extend `TransitionLabelMaker` (see Figure 2.18). Below we describe the implementation of three examples.

Let us consider the labelling of those states in which a specific method is invoked. The invocation of a static method corresponds to an `INVOKESTATIC` bytecode instruction. It seems intuitive to break the transition before the `INVOKESTATIC`, as we would like to label the state in which the static method will be invoked, as shown in Figure 2.19. The invocation of a non-static method corresponds to an `InstanceInvocation` bytecode instruction. Similarly, we would like to break the transition before the `InstanceInvocation`.

We create the following class `InvokedMethod` to define this labelling function.

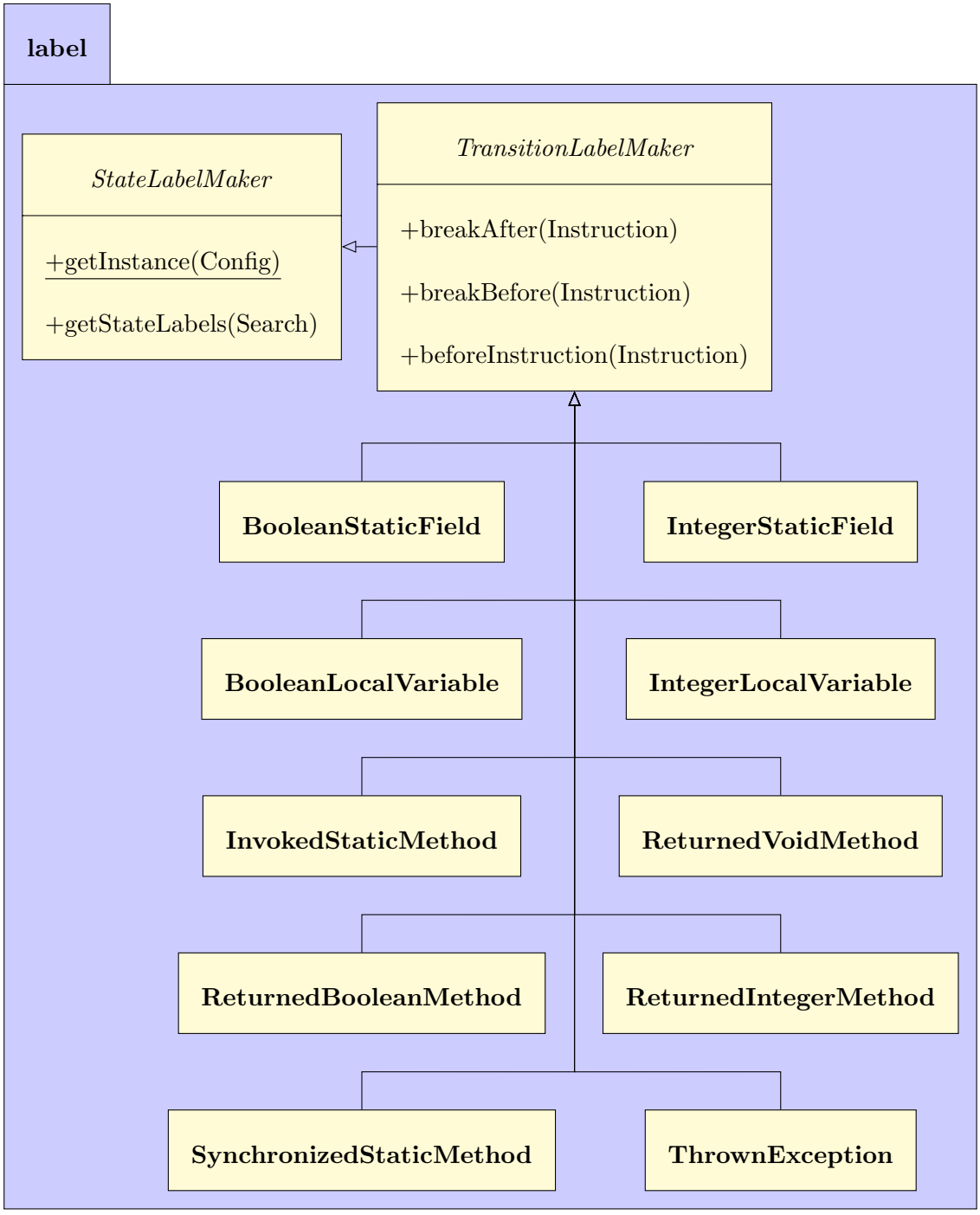


Figure 2.18: UML diagram of the transition labelling classes.

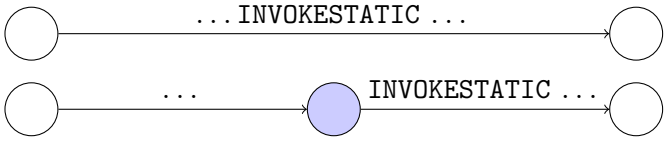


Figure 2.19: Transition is broken into two before `INVOKESTATIC` to label the invocation of a static method.

The class, which is a subclass of `TransitionLabelMaker`, contains the required `getInstance` method and overrides the `breakBefore` method, since we would like to break the transition before each relevant `INVOKESTATIC` or `InstanceInvocation` bytecode instruction.

```
1  package label;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  import gov.nasa.jpf.Config;
7  import gov.nasa.jpf.jvm.bytecode.INVOKESTATIC;
8  import gov.nasa.jpf.jvm.bytecode.InstanceInvocation;
9  import gov.nasa.jpf.util.MethodSpec;
10 import gov.nasa.jpf.vm.Instruction;
11 import gov.nasa.jpf.vm.MethodInfo;
12 import gov.nasa.jpf.vm.Types;
13
14 /**
15  * A labelling function for methods when they are invoked.
16  *
17  * The methods to be labelled can be specified in the
18  * application properties file by setting the property
19  * label.InvokedMethod.method. Method signatures must be in
20  * the format: package.class.methodName.
21  */
22 public class InvokedMethod extends TransitionLabelMaker {
23   private String[] methodName; // method signatures
24
```

```
25    /**
26     * Initializes this labelling function.
27     */
28    private InvokedMethod(Config configuration) {
29      methodName = getConfiguredProperty(configuration,
          ↪ "label.InvokedMethod.method");
30    }
31
32    /**
33     * Creates an InvokedMethod object.
34     *
35     * @param configuration JPF's configuration.
36     * @return an instance of this class.
37     */
38    public static InvokedMethod getInstance(Config
        ↪ configuration) {
39      return new InvokedMethod(configuration);
40    }
41
42    /**
43     * Whenever an instruction is executed, determines whether
44     * to break the current transition before the next
45     * instruction or not.
46     *
47     * @param nextInstruction next instruction which will be
48     *                        executed.
49     * @return the set of labels for the new state to break
50     *         the transition, null otherwise.
51     */
52    @Override
53    public Set<Label> breakBefore(Instruction nextInstruction)
54    {
55      if (nextInstruction instanceof INVOKESTATIC) {
56        INVOKESTATIC instruction = (INVOKESTATIC)
            ↪ nextInstruction;
57        // Each method in JPF is represented by a MethodInfo
58        // object.
59        MethodInfo methodInfo = instruction.getInvokedMethod();
60        for (String method : methodName) {
61          if (MethodSpec.createMethodSpec(method).
                ↪ matches(methodInfo)) {
62            Set<Label> labels = new HashSet<Label>();
```

```
63            String signature = methodInfo.getClassName().
                 ↪ replaceAll("[$.]", "_") + "_" +
                 ↪ methodInfo.getJNIName();
64            labels.add(new Label("invoked__" + signature,
                 ↪ method + " is invoked"));
65            return labels;
66          }
67        }
68      } else if (nextInstruction instanceof
         ↪ InstanceInvocation) {
69        InstanceInvocation instruction = (InstanceInvocation)
             ↪ nextInstruction;
70        // The MethodInfo object is not yet initialized for an
71        // instance invocation.
72        String invokedClass =
             ↪ instruction.getInvokedMethodClassName();
73        String invokedMethodName =
             ↪ instruction.getInvokedMethodName().split("\\(",
             ↪ 2)[0]; // remove parameter types
74        String invokedMethodSignature =
             ↪ instruction.getInvokedMethodSignature();
75        for (String method : methodName) {
76          if (MethodSpec.createMethodSpec(method).
               ↪ matches(invokedClass, invokedMethodName) &&
               ↪ matchSignatures(invokedMethodSignature,
               ↪ method)) {
77            Set<Label> labels = new HashSet<Label>();
78            String signature = invokedClass.replaceAll("[$.]",
                 ↪ "_") + "_" +
                 ↪ Types.getJNIMangledMethodName(null,
                 ↪ invokedMethodName, invokedMethodSignature);
79            labels.add(new Label("invoked__" + signature,
                 ↪ method + " is invoked"));
80            return labels;
81          }
82        }
83      }
84      return null;
85    }
86
87    /**
88     * Compares the signatures of the invoked method and a
```

```
89      * specified method to be labelled.
90      *
91      * @param invokedMethodSignature the signature of the
92      *                               invoked method.
93      * @param methodSignature        the signature of a
94      *                               specified method of
95      *                               interest.
96      * @return true if the signatures are equal, false
97      * otherwise.
98      */
99     private static boolean matchSignatures(String
           ↪ invokedMethodSignature, String methodSignature) {
100      String[] parameterTypes =
             ↪ methodSignature.split("\\s*[()]\\s*");
101      if (parameterTypes.length > 1) {
102        parameterTypes =
               ↪ parameterTypes[1].trim().split("\\s*,\\s*");
103        return Arrays.equals(parameterTypes, Types.
               ↪ getArgumentTypeNames(invokedMethodSignature));
104      }
105      // If the parameter types were not specified in the
106      // application properties file then label all methods
107      // with the specified class and method name.
108      return true;
109    }
110  }
```

The method `getInstance` on lines 38–40 returns an instance of `InvokedMethod` by calling the constructor on lines 28–30 and passing JPF's `Config` object as an argument. In the constructor, the signatures of the methods of interest are accessed from the configuration object in line 29. The user specifies the method signatures in the application properties file by setting the property `label.InvokedMethod.method` as demonstrated in Section 2.1.4. Note that specifying parameter types in the method signature is optional; however, if the method is overloaded and no param-

eter types are specified then all methods with the specified class and method name will be labelled.

In the method `breakBefore`, if the next instruction is an `INVOKESTATIC`, on line 59 we obtain the `MethodInfo` object of the static method to be invoked. If one of the methods specified in the application properties file matches the `MethodInfo` object, lines 62–65 are executed. In these lines we create the label `"invoked__<mangled method signature>"` and add it to the set of labels for the new state.

If the next instruction is an `InstanceInvocation`, the `MethodInfo` object of the non-static method to be invoked is not yet initialized. Thus, we must obtain the class signature, method name, and method signature separately, as in lines 72–74. If one of the methods specified in the application properties file matches this information, lines 77–80 are executed. In these lines we create the label `"invoked__<mangled method signature>"` and add it to the set of labels for the new state. Otherwise, we return `null`, signalling to continue the current transition.

Now let us consider the case where we would like to observe the return of a void method, which is represented by a `RETURN` bytecode instruction. It is intuitive to break the transition after the `RETURN`, as we can then label the state in which the void method has returned, as illustrated in Figure 2.20.

We create the following class which extends `TransitionLabelMaker`.

```
1 package label;
2
```

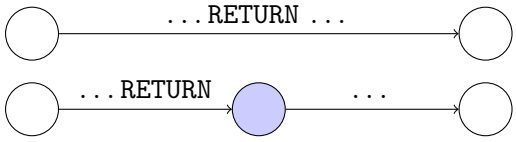


Figure 2.20: Transition is broken into two after `RETURN` to label the return of a void method.

```
3  import java.util.HashSet;
4  import java.util.Set;
5
6  import gov.nasa.jpf.Config;
7  import gov.nasa.jpf.jvm.bytecode.RETURN;
8  import gov.nasa.jpf.util.MethodSpec;
9  import gov.nasa.jpf.vm.Instruction;
10 import gov.nasa.jpf.vm.MethodInfo;
11
12 /**
13  * A labelling function for void methods when they return.
14  *
15  * The methods to be labelled can be specified in the
16  * application properties file by setting the property
17  * label.ReturnedVoidMethod.method. Method signatures must be
18  * in the format: package.class.methodName.
19  */
20 public class ReturnedVoidMethod extends TransitionLabelMaker
21 {
22   private String[] methodName; // method signatures
23
24   /**
25    * Initializes this labelling function.
26    */
27   private ReturnedVoidMethod(Config configuration) {
28     methodName = getConfiguredProperty(configuration,
         ↪ "label.ReturnedVoidMethod.method");
29   }
30
31   /**
32    * Creates a ReturnedVoidMethod object.
33    *
34    * @param configuration JPF's configuration.
```

```
35      * @return an instance of this class.
36      */
37     public static ReturnedVoidMethod getInstance(Config
           ↪ configuration) {
38       return new ReturnedVoidMethod(configuration);
39     }
40
41     /**
42      * Whenever an instruction is executed, determines whether
43      * to break the current transition after the executed
44      * instruction or not.
45      *
46      * @param executedInstruction the last instruction that
47      *                            was executed.
48      * @return the set of labels for the new state to
49      *         break the transition, null otherwise.
50      */
51     @Override
52     public Set<Label> breakAfter(Instruction
           ↪ executedInstruction) {
53       if (executedInstruction instanceof RETURN) {
54         RETURN instruction = (RETURN) executedInstruction;
55         MethodInfo methodInfo = instruction.getMethodInfo();
56         for (String method : methodName) {
57           if (MethodSpec.createMethodSpec(method).
                 ↪ matches(methodInfo)) {
58             Set<Label> labels = new HashSet<Label>();
59             String signature = methodInfo.getClassName().
                   ↪ replaceAll("[$.]", "_") + "_" +
                   ↪ methodInfo.getJNIName();
60             labels.add(new Label("returned__" + signature,
                   ↪ method + " returned"));
61             return labels;
62           }
63         }
64       }
65       return null;
66     }
67 }
```

The constructor and the method `getInstance` are very similar to those in the ex-

ample explained above. We override the method `breakAfter`, since we would like to break the transition immediately after a return of a specified void method. If the last executed bytecode instruction was a `RETURN`, we acquire the `MethodInfo` object in line 55. If one of the void methods specified in the application properties file matches the `MethodInfo` object, we return a set containing the label `"returned__<mangled method signature>"`. As a result of our framework, in this case the transition is broken and a new state is created which will be labelled. On the other hand, when the instruction is not a `RETURN` or a non-specified method has returned, we return `null` to continue the current transition.

Depending on the type of atomic property we want to track, the state labels may be more complex and the methods may require additional information from JPF such as the stack or the heap. For example, consider the labelling of states with the value of a static boolean field. We would need to obtain the value of the field from the heap. In addition, only when the value of the field is modified, we would need to break the transition immediately after the modification, in order to observe the effect of that modification. The assignment of a value to the field corresponds to a `PUTSTATIC` bytecode instruction. We only break the transition after the `PUTSTATIC`, as shown in Figure 2.3, if the value of the field has changed. In this way we refrain from introducing unnecessary states and we do not lose any valuable information either. The labelling function for this example is implemented as follows.

```
1  package label;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  import gov.nasa.jpf.Config;
7  import gov.nasa.jpf.jvm.bytecode.PUTSTATIC;
8  import gov.nasa.jpf.search.Search;
9  import gov.nasa.jpf.util.FieldSpec;
10 import gov.nasa.jpf.vm.ClassInfo;
11 import gov.nasa.jpf.vm.ClassLoaderInfo;
12 import gov.nasa.jpf.vm.ElementInfo;
13 import gov.nasa.jpf.vm.FieldInfo;
14 import gov.nasa.jpf.vm.Instruction;
15
16 /**
17  * A labelling function for static boolean attributes.
18  *
19  * The fields to be labelled can be specified in the
20  * application properties file by setting the property
21  * label.BooleanStaticField.field. Field signatures must be
22  * in the format: package.class.fieldName.
23  */
24 public class BooleanStaticField extends TransitionLabelMaker
25 {
26   private String[] fieldName; // field signatures
27   private Boolean previousValue; // of the last field to be
       ↪ modified
28
29   /**
30    * Initializes this labelling function.
31    */
32   private BooleanStaticField(Config configuration) {
33     fieldName = getConfiguredProperty(configuration,
         ↪ "label.BooleanStaticField.field");
34     previousValue = null;
35   }
36
37   /**
38    * Creates a BooleanStaticField object.
39    *
40    * @param configuration JPF's configuration.
```

```
41      * @return an instance of this class.
42      */
43     public static BooleanStaticField getInstance(Config
           ↪ configuration) {
44       return new BooleanStaticField(configuration);
45     }
46
47     /**
48      * Whenever the search advances to the next state, returns
49      * the labels associated with the new state.
50      *
51      * @param search JPF's search.
52      * @return the set of labels for the current state.
53      */
54     @Override
55     public Set<Label> getStateLabels(Search search) {
56       Set<Label> labels = new HashSet<Label>();
57       for (String field : fieldName) {
58         Boolean value = getValue(field);
59         if (value != null) {
60           labels.add(new Label(value + "__" +
                ↪ field.replaceAll("[$.]", "_"), field + " = " +
                ↪ value));
61         }
62       }
63       return labels;
64     }
65
66     /**
67      * Whenever an instruction is executed, determines whether
68      * to break the current transition after the executed
69      * instruction or not.
70      *
71      * @param executedInstruction the last instruction that
72      *                            was executed.
73      * @return the set of labels for the new state to
74      *         break the transition, null otherwise.
75      */
76     @Override
77     public Set<Label> breakAfter(Instruction
           ↪ executedInstruction) {
78       if (executedInstruction instanceof PUTSTATIC) {
```

```
79        PUTSTATIC instruction = (PUTSTATIC)
             ↪ executedInstruction;
80        FieldInfo fieldInfo = instruction.getFieldInfo();
81        // If the instruction modifies an attribute of
82        // interest then break the transition.
83        for (String field : fieldName) {
84          if (FieldSpec.createFieldSpec(field).
               ↪ matches(fieldInfo) &&
               ↪ !getValue(field).equals(previousValue)) {
85            return new HashSet<Label>();
86          }
87        }
88      }
89      return null;
90    }
91
92    /**
93     * This method is run whenever JPF's VM is about to
94     * execute the next instruction.
95     *
96     * @param instructionToExecute the next instruction to be
97     *                             executed.
98     */
99    @Override
100   public void beforeInstruction(Instruction
         ↪ instructionToExecute) {
101     if (instructionToExecute instanceof PUTSTATIC) {
102       PUTSTATIC instruction = (PUTSTATIC)
             ↪ instructionToExecute;
103       FieldInfo fieldInfo = instruction.getFieldInfo();
104       for (String field : fieldName) {
105         if (FieldSpec.createFieldSpec(field).
               ↪ matches(fieldInfo)) {
106           previousValue = getValue(field);
107         }
108       }
109     }
110   }
111
112   /**
113    * Returns the value of the given static boolean field.
114    *
```

```
115     * @param field the signature of the static boolean field.
116     * @return the value of the field if the class is
117     *         resolved, else null.
118     */
119    private Boolean getValue(String fieldSignature) {
120      ClassLoaderInfo loader =
           ↪ ClassLoaderInfo.getCurrentClassLoader();
121      int index = fieldSignature.lastIndexOf('.');
122      if (loader != null && index > 0) {
123        ClassInfo clazz = loader.tryGetResolvedClassInfo
             ↪ (fieldSignature.substring(0, index).trim());
124        if (clazz != null) {
125          FieldInfo field = clazz.getStaticField
               ↪ (fieldSignature.substring(index + 1).trim());
126          ElementInfo element = clazz.getStaticElementInfo();
127          if (element != null && field != null &&
               ↪ field.isBooleanField()) {
128            return element.getBooleanField(field);
129          }
130        }
131      }
132      return null;
133    }
134  }
```

The method `getInstance` on lines 43–45 returns an instance of the class `BooleanStaticField` by calling the constructor on lines 32–35 and passing JPF's `Config` object as an argument. In the constructor, the signatures of the fields of interest are accessed from the configuration file in line 33. The user specifies the field signatures by setting the property `label.BooleanStaticField.field` in the application properties file as demonstrated in Section 2.1.2.

The private method `getValue` on lines 119–133 returns the value of the boolean static field specified in the argument. We separate the field signature into the class

signature and the field name. Only if the class to which the field belongs can be resolved and the boolean field exists, the value will be returned. We retrieve the value from the heap using the `ElementInfo` object of the class, which contains the information of all static fields of the class, and the `FieldInfo` object of the given boolean static field. Otherwise, the method returns `null`.

We override the method `getStateLabels` on lines 54–64, as we would like to label all states with the values of the specified fields. Whenever a new state is reached, the method `getStateLabels` is executed. For each specified boolean static field, we create a label containing the value of the field with the field signature and add it to the set of labels for the new state.

Since we only break the transition if the value of a specified boolean static field has changed, we compare the value of the appropriate field before and after the `PUTSTATIC` instruction. To be able to carry out this comparison, we must store the value of the field to be modified before the `PUTSTATIC` instruction. Thus, we override the method `beforeInstruction`. If the instruction to be executed is a `PUTSTATIC` and a specified boolean static field will be modified, that is, the `FieldInfo` object matches one of the specified field signatures, we reach line 106. Here we assign the value of the specified field to the field `previousValue`.

In the overridden method `breakAfter`, if the last executed instruction was a `PUTSTATIC`, the `FieldInfo` object matches a specified field signature, and the field

was modified such that the current value is different from the previous value, we reach line 85. In line 85 we return an empty set of labels, because we want to break the transition but do not need to provide any labels, as the method `getStateLabels` will be executed and will return the full set of labels for the new state. If any of the three aforementioned conditions are not true, we return `null` in line 89 and we do not break the transition.

2.2.3 Labelling Format

As we already discussed in Section 2.1.1, the extension jpf-label currently supports two formats to output the state labelling, namely as a text file or as a dot file. To specify which output format JPF should use, the `listener` property should be set in the application properties file to the name of the class that implements the formatting.

jpf-label is versatile and we would like users to be able to apply the extension in many different scenarios. One such example is employing jpf-label in combination with PRISM to check probabilistic properties of Java code. As we will discuss next, jpf-label can easily be extended to support other output formats as well. In order to construct a new output format, we have to extend the abstract class `StateLabel`. This class is known as an event listener. It implements JPF's `SearchListener` and `VMListener` interfaces. These interfaces contain methods that correspond to

events. Whenever such an event occurs, JPF invokes the corresponding method. For example, whenever JPF's search of the state space finishes, JPF invokes the `searchFinished` method of all its registered listeners.

The class `StateLabel` keeps track of a list of all the labels that have been encountered during JPF's search of the state space. This list is captured by the field `allLabels`. Subclasses of `StateLabel` have direct access to this field, as we will see in an example presented below.

The new class not only needs to extend the abstract class `StateLabel`, but must also satisfy the following requirements.

1. It should contain a constructor that takes an instance of JPF's `Config` class as an argument. This constructor should first call the corresponding constructor of its superclass, that is, the `StateLabel` class. The `Config` object may be used to access properties defined in the application configuration file.

2. The abstract method `labelState` must be overridden. This method is executed whenever a new state is reached. The method `labelState` takes two arguments: an integer, representing the state id, and a set of integers, representing the indices of the labels of the state. For example, the arguments 3 and $\{1, 5, 6\}$ together represent that the state with id 3 has three labels, the indices of which are 1, 5 and 6. The purpose of this method is to handle

the labelling of each state in the desired output format. We will present an example shortly.

3. The abstract method `writeStateLabels` must be overridden. This method is executed whenever a search constraint is hit and also when JPF has completed its search of the state space. The method `writeStateLabels` takes two arguments: JPF's `Search` object and a string which contains the name of the system under test, appended with the search constraint if one was hit. The purpose of this method is to write the current labelling of the state space to a file.

4. Assume that the subclass overrides any of the methods `searchStarted`, `stateAdvanced`, `searchConstraintHit`, or `searchFinished` from the interface `SearchListener`. Then it should first call the corresponding method from the superclass `StateLabel`. Similarly, if the subclass overrides either of the methods `instructionExecuted` or `executeInstruction` from the interface `VMListener`, the corresponding method in the superclass should be invoked first.

For example, our extension offers the option to write the labelling of the states to a text file, in the format that is used by the model checker PRISM [KNP11]. The first line contains an enumeration of all the labels and their index, which is a non-negative integer. The remaining lines each contain the labelling of a state. This

is composed of the state followed by (the indices of) the labels of that state. Note that in JPF states are represented by either -1 or a non-negative integer. States that do not have any label are not included in the file. The generated file is named `<name of SUT>.lab`. We have created the following class to accomplish this.

```
1  package label;
2
3  import java.io.FileNotFoundException;
4  import java.io.PrintWriter;
5  import java.util.Set;
6
7  import gov.nasa.jpf.Config;
8  import gov.nasa.jpf.search.Search;
9
10 /**
11  * This listener outputs the labels of the state space to a
12  * file, named <name of SUT>.lab, in the format described
13  * below. All possible labels are enumerated by non-negative
14  * integers in the first line of the file. Subsequent lines
15  * capture the labelled states as follows: the state id
16  * followed by a colon and the indices of each of its labels,
17  * separated by a single space.
18  */
19 public class StateLabelText extends StateLabel {
20   private StringBuilder result;
21
22   /**
23    * Initializes the listener.
24    *
25    * @param configuration JPF's configuration.
26    */
27   public StateLabelText(Config configuration) {
28     super(configuration);
29     result = new StringBuilder();
30   }
31
32   /**
33    * Formats the labelling of the given state with the given
34    * set of labels.
```

```
35      *
36      * @param id     the id of the state.
37      * @param labels the set of indices of the labels.
38      */
39     @Override
40     public void labelState(int id, Set<Integer> labels) {
41       if (!labels.isEmpty()) {
42         result.append(id + ":");
43         for (Integer i : labels) {
44           result.append(" " + i);
45         }
46         result.append("\n");
47       }
48     }
49
50     /**
51      * Writes the current labelling of the state space to a
52      * file.
53      *
54      * @param search JPF's search
55      * @param name   the name of the system under test,
56      *               appended with the search constraint if one
57      *               was hit.
58      */
59     @Override
60     public void writeStateLabels(Search search, String name) {
61       try {
62         PrintWriter writer = new PrintWriter(name + ".lab");
63         writer.println(enumerateLabels());
64         writer.print(result);
65         writer.close();
66       } catch (FileNotFoundException e) {
67         System.out.println("Listener could not write to the
             ↪ output file " + name + ".lab");
68         search.terminate();
69       }
70     }
71
72     /**
73      * Enumerates the list of all labels.
74      *
75      * @return the string of enumerated labels
```

```
76     */
77   private String enumerateLabels() {
78     StringBuilder labelNames = new StringBuilder();
79     int n = allLabels.size();
80     for (int i = 0; i < n; i++) {
81       labelNames.append(i + "=\"" +
           ↪ allLabels.get(i).getName() + "\" ");
82     }
83     return labelNames.toString();
84   }
85 }
```

The above class contains a constructor that takes an instance of JPF's `Config` class as an argument and calls the corresponding constructor of its superclass. Moreover, the abstract methods `labelState` and `writeStateLabels` are overridden, thus, all of the above mentioned requirements are satisfied.

Recall that the method `labelState` is executed when a new state is reached and formats the labelling of the new state. If the state has at least one label, then the state id is appended to `result`, followed by a colon and the set of the indices of the labels separated by whitespace in lines 42–46. If the state has no labels, nothing is added to `result` regarding that state. In the class, we also override the method `writeStateLabels`. In line 62 the file named `<name of SUT>.lab` is created. The private method `enumerateLabels` on lines 77–84 collects the enumeration of all of the labels and their indices. In line 63, the representation of this collection is written to the file and in line 64, `result` is written to the file.

The extension jpf-label also includes the option to represent the labelled tran-

sition system graphically, that is, a file named `<name of SUT>.dot` is created, containing a directed graph with coloured vertices to represent the state labelling, and a file named `<name of SUT>_legend.dot` is created, containing the mapping between the state labels and colours. This is achieved by the listener `StateLabelDot` (see Figure 2.21). Examples of the use of these two listeners are discussed in Sections 2.1.1–2.1.6.

2.3 Implementation Details

The observer pattern is a software design pattern that defines a one-to-many dependency between objects [GHJV94]. In this pattern, an object maintains a list of its dependents and notifies them automatically of any state changes, usually by calling one of their methods. This object is known as the subject and its dependents are called observers. JPF uses the observer pattern to notify listeners when events occur. JPF's `SearchListener` and `VMListener` interfaces contain methods that correspond to events. Whenever such an event occurs, JPF invokes the corresponding method. For example, whenever JPF's search of the state space finishes, JPF invokes the `searchFinished` method of all its registered search listeners. In this case, JPF is the subject and the listeners are the observers.

The abstract class `StateLabel` is an event listener that implements both JPF's `SearchListener` and `VMListener` interfaces, as seen in Figure 2.21. `StateLabel`

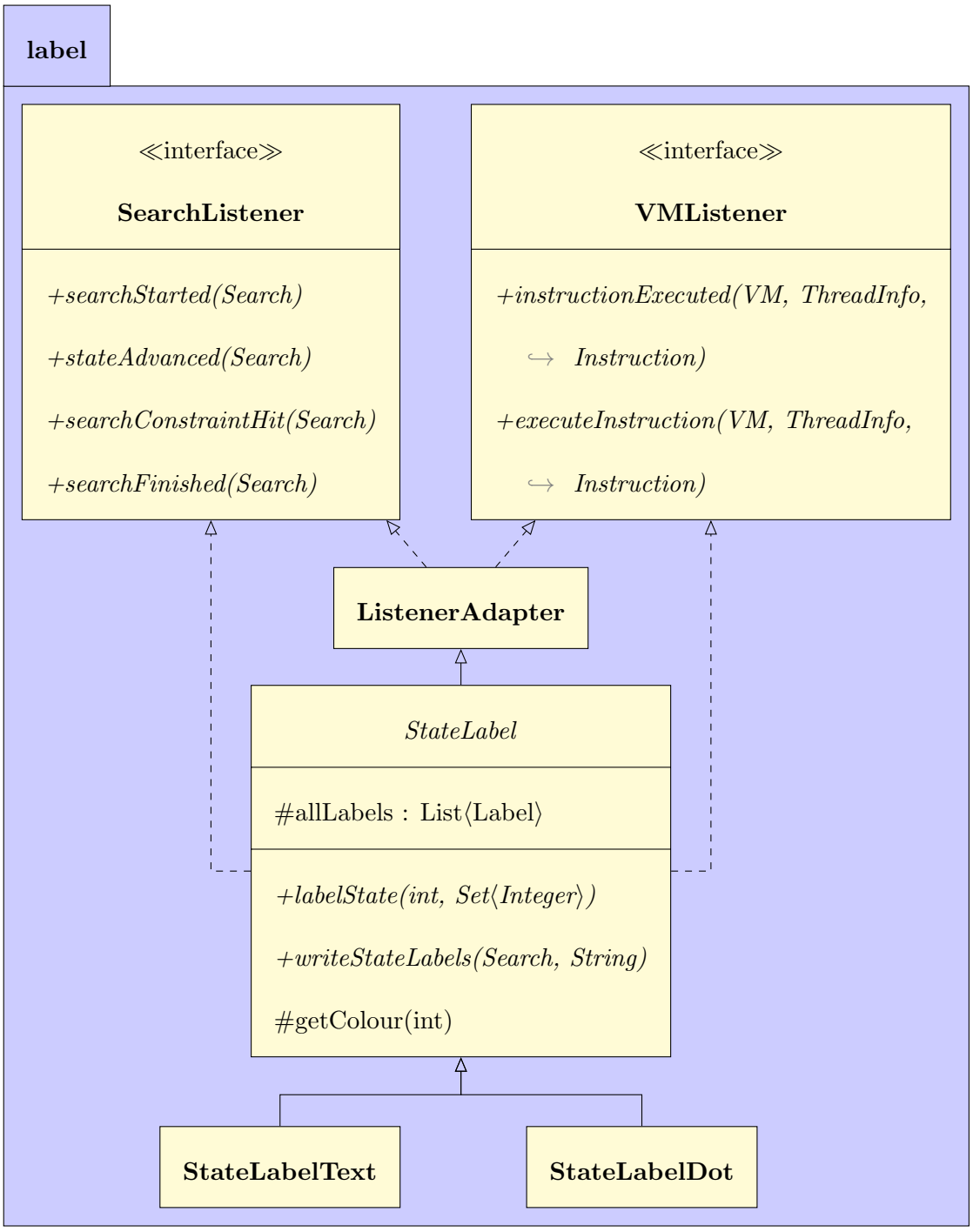


Figure 2.21: UML diagram of the listeners which specify the output format of the state labelling.

also extends the class `ListenerAdapter`, which is used to ease the implementation of listeners that only process a few notifications, by implementing all methods in the two aforementioned interfaces, with empty bodies. The purpose of the class `StateLabel` is to manage the various labelling classes, by allowing them to break transitions when certain events occur and by collecting the labels for each state when the state is reached. There is a one-to-many relationship between the class `StateLabel` and the labelling classes, therefore we have decided to use the observer pattern. Thus, `StateLabel` has a dual role of an observer of JPF and a subject with a list of dependent labelling classes. The simplified UML diagram in Figure 2.22 provides a visual representation of the classes in our extension and the relations between these classes.

`StateLabel` includes a list of labelling functions which were specified in the application properties file by the property `label.class`. These labelling functions are initialized using the `getInstance` method which must be defined in subclasses of `StateLabelMaker` (see Section 2.2.1).

When JPF's search begins, we are in the initial state -1 and JPF executes the method `searchStarted` in `StateLabel`. In turn, we execute the method `getStateLabels` in each of the labelling functions and gather the set of label indices for the initial state. Then we invoke the method `labelState` in `StateLabel` to label the initial state with its associated labels. Similarly, whenever JPF advances to the next

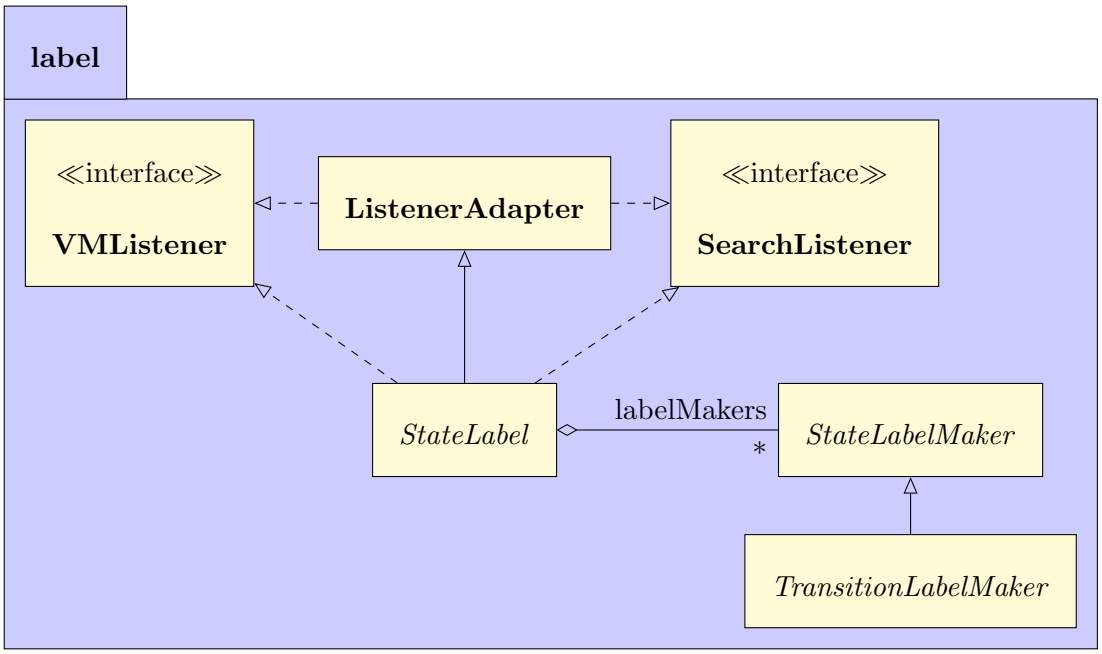


Figure 2.22: Simplified UML class diagram to show the use of the observer pattern in jpf-label.

state, the method `stateAdvanced` in `StateLabel` is executed. If the state is not identified with any previous state, that is, it is a new state, we execute the method `getStateLabels` in each of the labelling functions, collect the set of label indices for the current state and then label the state by invoking the method `labelState` with the result. Additionally, `StateLabel` stores a list of all the labels in the system.

Whenever JPF's virtual machine (VM) is about to execute an instruction, the method `executeInstruction` is run. In this method, we invoke the method `beforeInstruction` in all of the registered labelling classes that are subclasses of `TransitionLabelMaker`, so that the labelling classes may obtain any required

information. Each time the VM executes a bytecode instruction, the method `instructionExecuted` is run. In this method, we call the methods `breakAfter` and `breakBefore` in those labelling functions which are a `TransitionLabelMaker`. If any of the registered labelling classes indicate that the current transition must be broken, by returning labels for the generated state, we break the transition. When the transition is broken, a new state is created, which causes the method `stateAdvanced` to be executed. We then label this new state with all of the labels that have been returned by `breakAfter`, `breakBefore`, and `stateAdvanced`. Note that a transition may be broken either due to this listener or by JPF.

The class `StateLabel` writes the labelling to a file when a search constraint is hit or when JPF has completed its search of the state space, that is, when the methods `searchConstraintHit` and `searchFinished` are executed. `StateLabel` offers a template for outputting the labelling of the states, as discussed in Section 2.2.3. Additionally, `StateLabel` provides the protected methods `getColour`, which returns a unique colour per label index, and `generateLegendFile`, which produces a dot file containing a legend mapping each colour to the description of the label it represents. Its two subclasses, `StateLabelText` and `StateLabelDot`, are listeners that allow the user to represent the state labellings textually or graphically, respectively.

The abstract classes `StateLabelMaker` and `TransitionLabelMaker` provide a template for defining labelling functions as described in Section 2.2.1 and Section 2.2.2

respectively. A overview of the classes included in jpf-label and their purpose was summarized in Section 2.1.7. The classes `Initial` and `End` do not depend on the transitions in order to label states. These classes were explored in Section 2.2.1. The following labelling classes rely on breaking the transition before or after a specific bytecode instruction to correctly label states. For each labelling class, we will briefly discuss which bytecode instructions are relevant and where to obtain any additional required information. For more detail, see Section 2.2.2.

- `BooleanStaticField`: The assignment of a value to a static field corresponds to a `PUTSTATIC` bytecode instruction. The transition is broken after the `PUTSTATIC`, if the value of the field has been modified, to observe the effect of the assignment to the field, as shown in Section 2.2.2.

  This class can be modified to handle static fields with other types, by using the appropriate getter for the field type when returning the value of the field. In order create labels for non-static fields, consider the bytecode instruction `PUTFIELD` rather than `PUTSTATIC` to break the transitions. Moreover, information regarding non-static fields is stored in the `DynamicElementInfo` of the class instead of the `StaticElementInfo`.

- `IntegerLocalVariable`: The assignment of an integer value to a variable corresponds to an `ISTORE` bytecode instruction, while incrementing or decrementing

an integer variable corresponds to an `IINC` bytecode instruction. The transition is broken after an `ISTORE` or `IINC` if the variable's value is modified. The current value of a variable can be obtained from the stack. However, if the bytecode instruction was the last instruction in the scope of the local variable, the variable will no longer be on the stack, thus we must save the last value of a local variable.

Local boolean variables are represented as integers, that is, the value `true` is represented by a 1 and the value `false` is represented by a 0, thus assignments to boolean variables correspond to `ISTORE` bytecode instructions as well. To label local variables of other types, that is double, float, long or non-primitive types, inspect the bytecode instructions `DSTORE`, `FSTORE`, `LSTORE`, or `ASTORE`, respectively. When obtaining the value of local variables of type double or long, the method `getLocalVariable` should be replaced with the method `getLongLocalVariable`.

- `InvokedMethod`: The invocation of a static method is represented by the bytecode instruction `INVOKESTATIC`, while the invocation of a non-static method is represented by the bytecode instruction `InstanceInvocation` or any of its subclasses. We break the transition before the `INVOKESTATIC` or `InstanceInvocation` so that we can label the state in which the method will be invoked, as de-

scribed in Section 2.2.2.

- `ReturnedVoidMethod`: The return of a void method corresponds to a `RETURN` bytecode instruction. We break the transition after the `RETURN`, as we can then label the state in which the void method has returned, as described in Section 2.2.2.

- `ReturnedBooleanMethod`: The return of a boolean method corresponds to an `IRETURN` bytecode instruction. Similarly, we break the transition after the `IRETURN`, as we can then label the state in which the boolean method has returned. The return value is accessed with the method `getReturnValue`.

  This can be modified for methods with other return types, such as float, int, double, long or non-primitive types by checking the bytecode instructions `FRETURN`, `IRETURN`, `DRETURN`, `LRETURN`, or `ARETURN`, respectively. The return of a native method corresponds to a `NATIVERETURN` bytecode instruction.

- `SynchronizedStaticMethod`: We label those states in which the synchronized static method acquires and has released the lock. Therefore, we break the transition before the invocation of the static method (when the method acquires the lock), which corresponds to an `INVOKESTATIC` bytecode instruction. We also break the transition after the return of the method (when the method has released the lock). Since we accommodate for all return types, we use

the general bytecode instruction `ReturnInstruction`. Furthermore, we must ensure that the method has the synchronized modifier, which is represented by the number 32.

To label non-static synchronized methods, we may check the bytecode instruction `InstanceInvocation` or any of its subclasses.

- `ThrownException`: An exception or error which is explicitly thrown corresponds to an `ATHROW` bytecode instruction. We break the transition after the `ATHROW` to label those states in which an exception/error has been thrown. However, an `ATHROW` bytecode instruction removes the information about the exception/error from the stack, thus we need to store all required information before the `ATHROW`.

2.4 Summary

jpf-label expands the functionality of the model checker JPF, by providing an easy way to label the states of JPF's virtual machine with a set of atomic propositions. Our extension already supports a range of atomic propositions, which express simple known facts about the states. 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 is invoked or returns, etc. Moreover, the extension

has been designed in such a way that it can be easily extended. Thus, users can conveniently define their own labelling functions to label states with desired atomic propositions. Our extension supports both state labelling and transition labelling.

jpf-label currently supports two formats to output the state labelling, namely as a text file in the format used by the model checker PRISM or as a dot file containing a graphical representation of the labelled state space as a coloured directed graph with a legend. The extension also enables users to readily construct their own format for the output of the labelling.

3 Probabilistic Model Checking of Java Code

jpf-probabilistic, an extension of JPF, assigns probabilities to the transitions, which reflect the random choices in the Java code [ZvB10]. For the extension to detect the random choices in Java, they must be expressed using one of the four Java classes included in jpf-probabilistic. The class `Choice` contains the method `make` which takes an array of `doubles`, say `p`, as an 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 $\texttt{p}[i]$. The classes `Coin` and `Die` provide the methods `flip` and `roll`. The method invocation `UniformChoice.make`(n) returns i, where $0 \leq i < n$, with probability $\frac{1}{n}$. By adding probabilities to the transitions, jpf-probabilistic turns a transition system into a (discrete time) Markov chain.

When we run JPF extended with jpf-probabilistic on Java code, we can generate a file that contains the graphical representation of the Markov chain corresponding to the code. Let us use the following Java app as an example.

```
1 import probabilistic.Coin;
2
```

```
3  public class Probabilistic {
4    private static boolean value = true;
5
6    public static void main(String[] args) {
7      if (Coin.flip() == 1) {
8        Probabilistic.value = false;
9        Probabilistic.value = true;
10     } else {
11       Probabilistic.value = (Coin.flip() == 1);
12     }
13   }
14 }
```

Observe that we use the method `flip`, of the class `Coin` described above, to choose either `0` or `1` with equal probability. To obtain the Markov chain corresponding to this Java app, we create the following application properties file.

```
1 target = Probabilistic
2 classpath = <path to the directory containing
    ↪ Probabilistic.class>
3
4 @using jpf-probabilistic
5 listener = probabilistic.listener.StateSpaceDot
```

The system under test (SUT), that is, the Java app to be model checked, is given in line 1. The directory that contains the bytecode of the app needs to be added to JPF's classpath. This is done in line 2. Line 4 specifies that we use JPF's extension jpf-probabilistic. To specify that JPF should provide a graphical representation of the state space with probabilities, we set the `listener` property in line 5 to the class `StateSpaceDot`. Running JPF with these application properties results in the Markov chain depicted in Figure 3.1. Note that the prob-

abilities are displayed with two decimal places, which is the default. The property `probabilistic.listener.StateSpaceDot.precision` is optional and captures the precision of the probabilities in the graphical representation of the state space. For example, if we want the probabilities provided up to three decimal places, then we set the above property to 3.

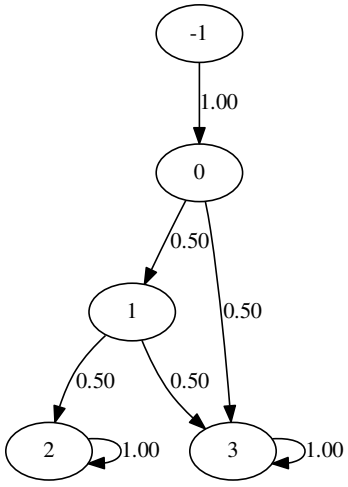


Figure 3.1: The graphical representation produced by jpf-probabilistic.

The transition from the initial state -1 to state 0 corresponds to the sequence of bytecode instructions that includes the initialization of the field `value`, the invocation of the `main` method, and the execution of this method up to `Coin.flip()` in line 7. This portion of the code is deterministic and will always be executed, thus the transition has a probability of `1.00`.

The two transitions that leave state 0 correspond to the two possible results of `Coin.flip()` on line 7. State 1 is reached if the result is `0`, that is, if line 11 is

reached. Since the method returns `0` and `1` with equal probability, this transition has a probability of `0.50`. The two outgoing transitions of state 1 represent the possible results of `Coin.flip()` on line 11. If the result is `0` then the field `value` is false and final state 2 is reached. The transition between state 1 and state 2 has a probability of `0.50` to be taken. If the result in line 11 is `1` then the final state 3 is reached and the transition between state 1 and state 3 also has a probability of `0.50`. Since state 2 and state 3 are final states, the probability of remaining in those states is `1.00`.

Finally, we consider the other transition from state 0 that corresponds to a result of `1` in line 7, leading to the execution of lines 8–9. This results in a final state in which the field `value` is true, that is, state 3. Hence the probability associated with the transition between state 0 and state 3 is `0.50`.

jpf-probabilistic can also write the Markov chain to a transition file. To specify this, we replace line 5 of the above application properties file with

```
5 listener = probabilistic.listener.StateSpaceText
```

which results in the following output written to the file `Probabilistic.tra`. Note that we do not limit the precision of the probabilities in the transition file.

```
1 5 7
2 -1 0 1.0
3 0 1 0.5
4 1 2 0.5
5 2 2 1.0
```

```
1 3 0.5
3 3 1.0
0 3 0.5
```

The first line contains the number of states and the number of transitions. The remaining lines describe the transitions. Each line contains the source state, the target state and the probability of the transition from the source state to the target state.

3.1 Using jpf-label with jpf-probabilistic

When extended by both jpf-label and jpf-probabilistic, JPF can produce a graphical representation of the labelled Markov chain as a directed graph where the edges are associated with probabilities and the vertices are coloured according to their labels. This is accomplished by the following two listeners which are part of the package `probabilistic.listener` in jpf-probabilistic.

- `StateSpaceDot`: writes a graphical representation of the Markov chain to a file.
- `StateLabelVisitor`: assigns colours to the states according to their labels, in the same file created by `StateSpaceDot`. Note that this listener also supports the classes to label states discussed in Section 2.1.7.

To demonstrate how to use jpf-label in conjunction with jpf-probabilistic, we

will adjust the application properties file as indicated below.

```
1 target = Probabilistic
2 classpath = <path to the directory containing
    ↪ Probabilistic.class>
3
4 @using jpf-label
5 @using jpf-probabilistic
6 listener = probabilistic.listener.StateSpaceDot;
    ↪ probabilistic.listener.StateLabelVisitor
7 label.class = label.End
```

We add line 4 to specify that we are using JPF's extension jpf-label, in addition to the extension jpf-probabilistic. To specify that JPF should provide a graphical representation of the state space with labels and probabilities, we set the `listener` property in line 6 to both the classes `StateSpaceDot` and `StateLabelVisitor`. The property `label.class` captures which states are labelled. In line 7 we specify that the final states are labelled. When we run JPF on this application properties file, we obtain the labelled Markov chain displayed in Figure 3.2.

JPF, extended by jpf-label and jpf-probabilistic, is also able to construct a textual representation of this labelled Markov chain, if we modify line 6 of the application properties file as follows.

```
6 listener = probabilistic.listener.StateSpaceText;
    ↪ label.StateLabelText
```

This results in the creation of two files, that is, a transition file and a labelling file. The transition file is called `Probabilistic.tra` and was discussed earlier in

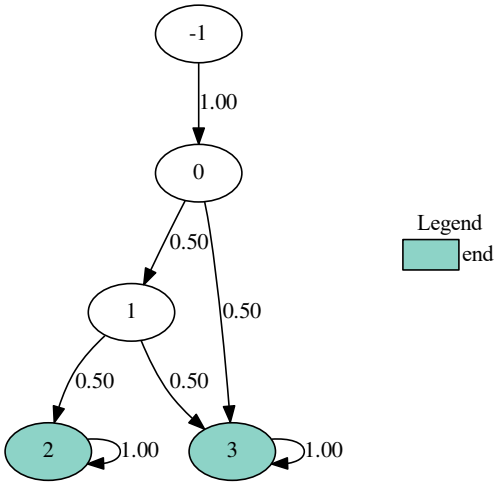


Figure 3.2: Labelling of the final states of a Markov chain.

this chapter. The labelling file is called `Probabilistic.lab` and has the following contents.

```
0="end"
2: 0
3: 0
```

3.2 The Bridge Between JPF and PRISM

PRISM [KNP11] 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. Furthermore, it takes a probabilistic property specified in a logic as input. PRISM checks, among other things, whether the model satisfies the property.

As discussed in Section 3.1, the extensions jpf-label and jpf-probabilistic provide

a way to extract the underlying labelled Markov chain of a Java application. The labels capture a set of atomic propositions about the states of JPF's virtual machine, which may be used to express properties of the code. Such properties can be formalized in logics such as linear temporal logic (LTL) [Pnu77]. The transition and labelling files produced by JPF, with the help of jpf-label and jpf-probabilistic, can be converted into a format that can be fed into PRISM together with a probabilistic property.

The format of the transition and labelling files generated by JPF differs slightly from PRISM's input format. JPF numbers its states using consecutive integers beginning at -1, whereas PRISM starts from zero. JPF may produce multiple transitions between a given pair of states, whereas 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 with `"init"`. Therefore, we have implemented a simple converter, named `JPFtoPRISM`, that renumbers the states in the transition and label files. The converter checks if all labels satisfy the above mentioned restrictions and if the initial state is not labelled, the converter adds the label `"init"` to the initial state of the model. If JPF has produced multiple transitions from a given source to a given

target, then the converter collapses those transitions into a single transition between the source and the target by adding up the transition probabilities. Moreover, the converter checks that the probabilities of the outgoing transitions of each state sum to one and adds a labelled sink state for the remaining probability, if necessary. This ensures that if JPF has not traversed the state space completely, for example, because it ran out of memory, then the resulting Markov chain's transition matrix is a right stochastic matrix, and, as a result, we do not get a deadlock warning in PRISM. Note that running out of memory is an unfortunate problem that occurs rather frequently when using model checking.

To illustrate how to use JPF in tandem with PRISM, we will continue using the example introduced earlier in this chapter. Firstly, we translate JPF's output into PRISM's input format using the converter with the following command.

```
1 java probabilistic.tool.JPFtoPRISM Probabilistic PRISMModel
```

The converter takes in two command line arguments. The first argument specifies the file name of the Markov chain generated by JPF, that is, `Probabilistic.tra` and `Probabilistic.lab`, which were discussed in Section 3.1. The second argument specifies the desired PRISM model file name. The converter produces two files. The transition file `PRISMModel.tra` has the following content.

```
1 5 7
2 0 1 1.0
3 1 2 0.5
```

```
1 4 0.5
2 3 0.5
2 4 0.5
3 3 1.0
4 4 1.0
```

The ID of each state is increased by one, but the transitions and their probabilities remain the same. The contents of the labelling file `PRISMModel.lab` is shown below. Similarly, the ID of each state is increased by one; however, the converter also adds the extra label `init` to the intial state 0.

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

Lastly, we use PRISM to compute properties for this labelled Markov chain. Consider the following properties file named `Properties.pctl`.

```
P>=1 [ F "end" ];
P=? [ X X "end" ];
```

The first property `P>=1 [ F "end" ]` indicates that PRISM should check that the property `F "end"` holds with probability 1. The property specifies that eventually a state labelled `end` is reached, that is, the algorithm eventually terminates successfully. The second property, `P=? [ X X "end" ]` states that PRISM should compute the probability that the property `X X "end"` holds. The property specifies that the next next state is labelled `end`, that is, the algorithm terminates in two steps.

We supply the labelled Markov chain and the properties to PRISM with the following command, indicating that the model is a discrete time Markov chain.

```
prism -importmodel PRISMModel.tra,lab -dtmc Properties.pctl
```

PRISM then returns the results of the model checking. For the first property `P>=1 [ F "end" ]`, PRISM returns `true`, verifying that the property holds in the model. For the second property `P=? [ X X "end" ]`, the outcome is the numerical value `0.5`, which corresponds to the probability of transitioning from the initial state 0 to state 1 and then to the final state 4.

We will discuss an example where ghost variables are needed in order to specify a desired property in Section 4.2. For an example in which transitions to a sink state are added, see Section 4.3.

In summary, the extensions of JPF, jpf-label and jpf-probabilistic, expand the functionality of the model checker. Both extensions have been designed in such a way that they themselves can be easily extended. Together with our converter, they build a bridge between the model checkers JPF and PRISM. We now can check properties expressed in logics such as LTL and PCTL of randomized Java code. As far as we know, this provides the first model checking tool, depicted in Figure 3.3, that can check probabilistic properties of Java code. Furthermore, we can use PRISM to supplement JPF's qualitative results with quantitative information as will be shown with more practical examples in Chapter 4.

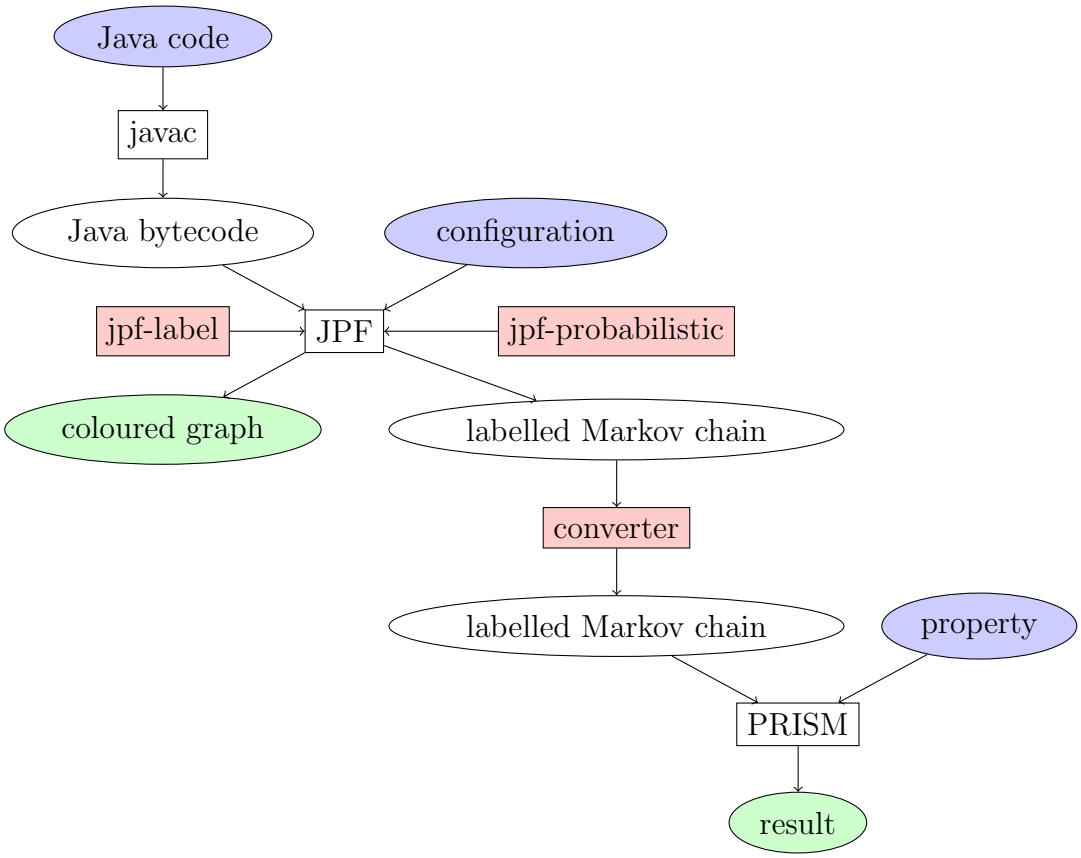


Figure 3.3: The diagram provides an overview of the model checking tool. The ovals are data and the rectangles are tools. The blue ovals are input. The green ovals are output. The red rectangles are the parts that I developed (jpf-label and the converter) or added to (jpf-probabilistic).

3.3 Implementation Details

The implementation of jpf-probabilistic is discussed in detail by Zhang in [Zha10]. In this section, we will only study the implementation of the classes that we have added to jpf-probabilistic, which allow us to use jpf-probabilistic with jpf-label. We will also review the converter that translates JPF's output into PRISM's input.

3.3.1 jpf-probabilistic

jpf-probabilistic extracts the underlying Markov chain from Java code and assigns probabilities to its transitions. Currently, the Markov chain can be returned in two formats, namely textual and graphical. This is achieved by the listeners `StateSpaceText` and `StateSpaceDot`, respectively.

We would like to add the state labelling provided by jpf-label to the graphical representation of the Markov chain, by colouring the states according to their labels. Since this functionality is unrelated to jpf-probabilistic, we do not want to modify the listener `StateSpaceDot`. Thus, we implement the visitor pattern [GHJV94], which separates the labelling from jpf-probabilistic and also allows the user to add new operations easily. We add the `Visitor` interface to jpf-probabilistic as shown in Figure 3.4. In the `StateSpaceText` class, the method `accept` invokes the `Visitor`'s method `visitStateSpaceText`, while in the `StateSpaceDot` class, the method `accept` invokes the `Visitor`'s method `visitStateSpaceDot`.

We then create the class `StateLabelVisitor` which implements the interface `Visitor` and extends the abstract class `StateLabel` from the extension jpf-label[7]. Following the guidelines in Section 2.2.3, we include a constructor that takes an instance of JPF's `Config` class as an argument and calls the superconstructor. We then implement the following two abstract methods of `StateLabel`.

[7]The class `StateLabelVisitor` is only compiled if jpf-label has been installed already.

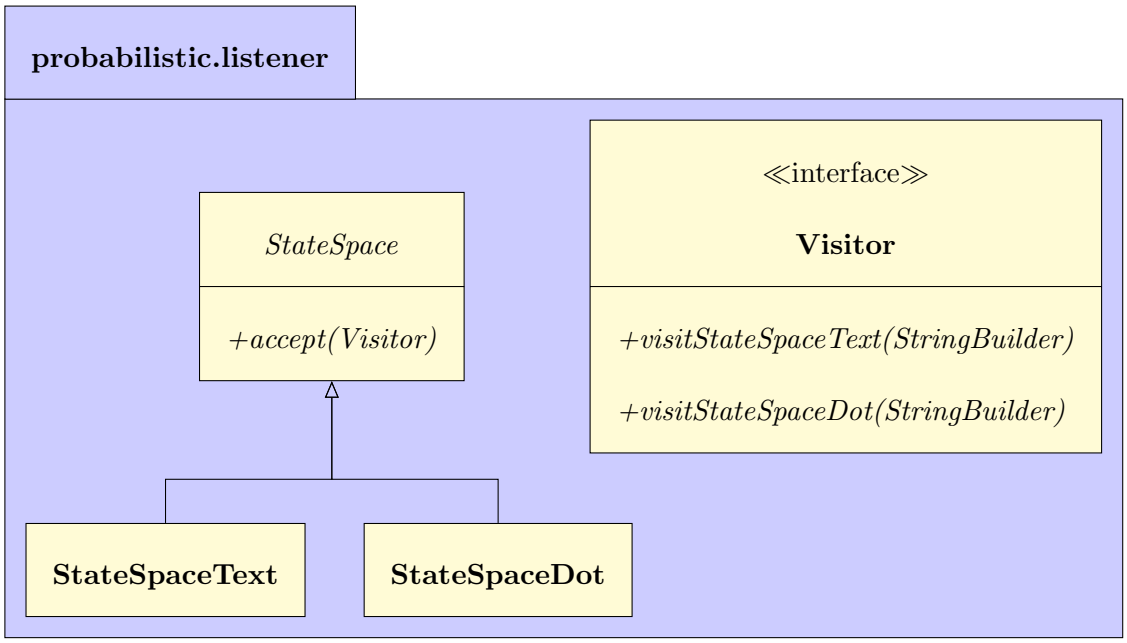


Figure 3.4: A UML diagram of the use of the visitor pattern in jpf-probabilistic.

The first abstract method `writeStateLabels` is executed when a search constraint is hit and when JPF's search terminates. This method handles the writing of the labelling to a file. However, since we want to add the labelling to the file created by `StateSpaceDot` and not a separate file, we leave the body of this method blank.

The second abstract method `labelState` is executed whenever a new state is reached and takes two arguments, namely the state ID and a set of label IDs. Using the method `getNextListenerOfType` in JPF's `Search` class, we can traverse through all listeners of type `StateSpaceDot`. The visitor class invokes the method `accept` in each of these listeners, passing as an argument a reference to itself.

In turn, the `accept` method of the class `StateSpaceDot` invokes the method `visitStateSpaceDot` in the `StateLabelVisitor` class, passing as an argument the `StringBuilder` object that will be written to the output file. In the method `visitStateSpaceDot` we decorate the current state by colouring it according to its set of labels. This is achieved by appending commands in the dot language to the provided `StringBuilder`. For example, if state 7 had three labels with the indices 1, 5 and 6, we would express this in the dot language as the following.

```
1 7 [fillcolor="1:5:6"]
```

Lastly, since we do not wish to include the state labelling in the transition file, we leave the method `visitStateSpaceText` empty.

Let us now consider the case where JPF runs out of time or memory and we would like to determine which parts of the state space have been fully explored. We create a visitor `ExploredStatesVisitor` to mark those states that have been fully explored by the VM. The class `ExploredStatesVisitor` implements the interface `Visitor` and extends the class `ListenerAdapter`.

We override two methods from the class `ListenerAdapter`, namely `searchStarted` and `stateAdvanced`. The method `searchStarted` is executed when JPF's search begins. In this method we initialize the field `id`, which will be used to keep track of the current state ID. We also invoke the method `accept` in each listener of type `StateSpaceDot`, since we would like to mark the initial state as it has been fully

explored. The method `stateAdvanced` is executed whenever JPF's search advances to the next state. In this method, we update the field `id` to the ID of the current state. If the choice generator of the current state has no more choices or if the current state is an end state, the state has been fully explored, thus we invoke the method `accept` in each listener of type `StateSpaceDot`.

In the method `visitStateSpaceDot`, we mark explored states by adding a second circle around them. We accomplish this using the following dotty command, where `<state id>` is replaced with the integer representing the ID of the current state.

```
1 <state id> [peripheries=2]
```

Since we would only like to mark explored states in the graphical representation of the Markov chain and not in the transition file, we leave the method `visitStateSpaceText` empty.

The UML diagram in Figure 3.5 illustrates the relationships between the visitor classes described in this section. The user can add additional information to the textual or graphical representation of the Markov chain by implementing the interface `Visitor`. To specify which listener(s) and (optionally) visitor(s) JPF should use, the property `listener` in the application properties file should be set to the desired class signatures.

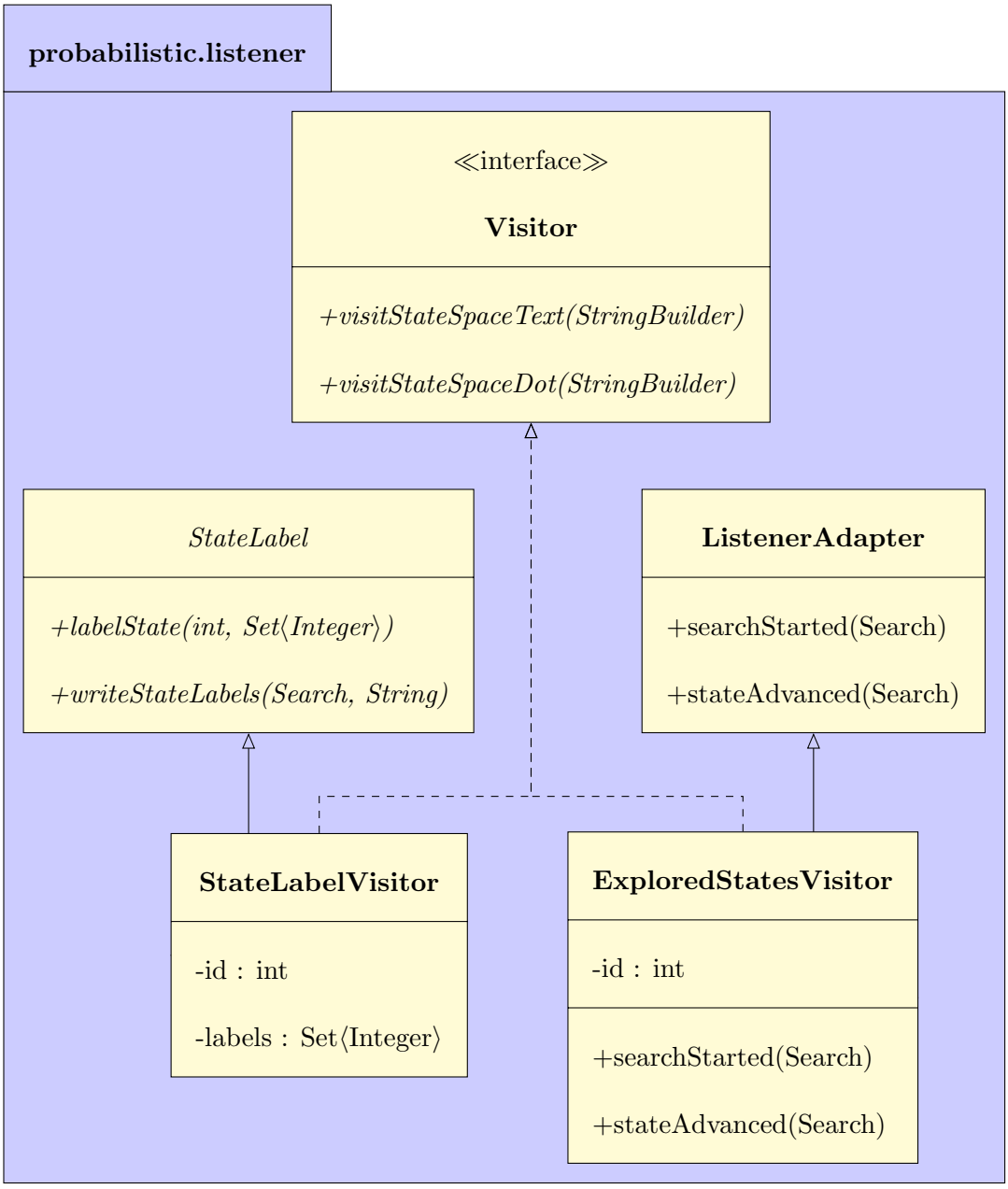


Figure 3.5: A UML diagram of the visitor classes in jpf-probabilistic.

3.3.2 JPFtoPRISM

The converter `JPFtoPRISM` is included in the package `probabilistic.tool` of the extension jpf-probabilistic. The converter takes two command line arguments. The first argument specifies the file name of the Markov chain generated by JPF, that is, the input. The second argument specifies the desired PRISM model file name, that is, the output.

First, we read the contents of JPF's transition file. From the first line, we obtain the total number of states and transitions. From each subsequent line, we get a transition's source state ID, target state ID and probability. We increase all state IDs by 1, so that they may be consecutive non-negative integers beginning at 0. For each state, we create a map of its outgoing transitions, with the target state ID as the key and the total probability of going from this source state to the target state as the value. Thus, we collect transitions so that there is at most one transition between each source-target pair. For each state, we also keep track of the sum of the probabilities of its outgoing transitions. Once we have read every transition from the transition file, if any state has a total outgoing probability of less than one, with a relative difference greater than 10^{-12}, we add a sink state. Then for each state with a total outgoing probability of less than one, we add a transition from that state to the sink state with the remaining probability and update the

total number of transitions. Hence, completing the Markov chain and ensuring that the Markov chain does not have any deadlocked states. We then write the total number of states and transitions to the new transition file, followed by the transitions. Each transition is represented by its source state ID, target state ID and probability, separated by whitespace and followed by a newline.

Second, we read the contents of JPF's label file. The first line contains an enumeration of all the labels and their index, which is a non-negative integer. We check that the following PRISM restrictions and requirements are satisfied.

- A label name should not be a reserved keyword in PRISM. If any of the label names are one of PRISM's keywords (see Appendix B), we throw the custom exception `IncorrectFileFormatException`.

- In PRISM, a label may only consist of letters, digits and the underscore character. Labels can neither begin with a digit nor contain any whitespace. The regular expression `[A-Za-z_][A-Za-z0-9_]*` captures this requirement. If a label does not match this regex, we throw the custom exception `IncorrectFileFormatException`.

- PRISM requires that the initial states of the model are labelled with `"init"`. If this label is present in the label file, we issue a warning to the user that states labelled with this label will be considered as initial states. If this label

is not present in the label file, we add the label `"init"` to the initial state 0.

Furthermore, if a sink state was added while creating the new transition file, we label that sink state with the label `"sink"`. If the label `"sink"` is already present in the label file, we issue a warning to the user that the label may not be unique to the sink state. We then create the new label file and write to it the enumeration of the labels with their index. The remaining lines of the input label file each contain the labelling of a state. This is composed of the state ID followed by (the indices of) the labels of that state. States that do not have any label are not included in the file. As with the transition file, we increase each state ID by one and write the state labelling to the newly created label file.

3.4 Summary

jpf-probabilistic, an extension of JPF, assigns probabilities to the transitions, which reflect the random choices in the Java code, turning the transition system into a discrete time Markov chain. My additions to jpf-probabilistic allow us to add the state labelling provided by jpf-label to the graphical representation of the Markov chain, by colouring the states according to their labels. Moreover, the visitor framework allows a user to easily add additional information to the textual or graphical representation of the Markov chain.

My converter `JPFtoPRISM`, which is included in jpf-probabilistic, transforms the

transition and labelling files generated by JPF into PRISM's input format. Thus, jpf-probabilistic and jpf-label, together with the converter, enable us to use the model checkers JPF and PRISM in tandem. We now can check properties expressed in logics such as LTL and PCTL of randomized Java code. This provides the first model checking tool that can check probabilistic properties of randomized algorithms implemented in a modern programming language.

4 Examples

To illustrate how our tool can be used, we present several randomized algorithms implemented in Java. For each algorithm, we define a property we would like to investigate and choose appropriate labelling functions to label the states accordingly. We extract the underlying labelled Markov chain of the Java code using JPF extended with jpf-label and jpf-probabilistic. We then check the probabilistic property with PRISM and discuss the results.

This chapter is based on collaborative research with Xiang Chen, Yash Dhamija, and Maeve Wildes. More specifically, the examples presented in Sections 4.1 and 4.3 were implemented by me. The algorithm presented in Section 4.2 was implemented by Xin Zhang and the algorithms presented in Sections 4.4 and 4.5 were implemented by Yash Dhamija, while the properties for these three examples were defined by Xiang Chen. Note that all of these randomized examples are different from those provided or analyzed by other tools (see Appendix C).

4.1 Primality Test

The Miller-Rabin primality test [Rab80] determines whether a number given as input is prime. This algorithm has been implemented in the method `isProbablePrime` of the class `java.math.BigInteger`. This is a Monte Carlo algorithm as it may erroneously report with a small probability that the number provided as an argument is prime. The algorithm contains a main loop, as shown in Algorithm 1. The more iterations of this loop executed, the lower the probability that the algorithm returns an incorrect result. We will show that our tool can compute the probability of this algorithm incorrectly reporting that a composite number is prime.

Algorithm 1: Miller-Rabin(n, k)

Input: $n \in \mathbb{N}$ an integer greater than 1
$k \in \mathbb{N}$ the number of iterations
Output: *false* if n is found to be composite, *true* otherwise

```
1 repeat k times
2     compute r and R such that n − 1 = 2^r R and R is odd
3     choose a uniformly at random from [1, n − 1]
4     for i ← 0 to r do b_i ← a^(2^i R)
5     if b_r mod n ≠ 1 then return false
6     if b_0 mod n = 1 then return true
7     j ← max{i | b_i mod n ≠ 1}
8     return b_j mod n = n − 1
9 end
```

We have implemented the algorithm in Java in a method called `isPrime` of a class named `MillerRabinPrimalityTest`. The randomization in line 3 is captured by jpf-probabilistic's `UniformChoice.make` method. We will label the return of the

boolean method `isPrime`, so that we can determine the probability with which the method returns an incorrect result. We create the following application properties file.

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

Line 1 specifies that the Java app named `MillerRabinPrimalityTest` is to be model checked by JPF. In line 2, we provide the command line arguments, namely `9`, which is the number to be tested for primality, and `2`, which is the number of iterations of the main loop to be executed. The `classpath` tells JPF where to find the bytecode of the Java app. Lines 5 and 9 specify that our extensions jpf-label and jpf-probabilistic are used. Line 6 specifies that the initial state and the final states should be labelled, as well as those states in which the boolean method `isPrime` returns, as specified by the method signature in line 7. Finally, line 10 specifies that JPF should generate a labelled graphical representation of the state space and

line 11 captures that the probabilities of the transitions should be depicted with three digits precision.

Running JPF with this configuration file results in the creation of the file named `MillerRabinPrimalityTest.dot`, containing the labelled Markov chain depicted in Figure 4.1. The initial state -1 and the final states 3 and 5 are labelled, as well as those states in which the boolean method that determines whether the number is prime returns true (states 2 and 6) and false (state 4).

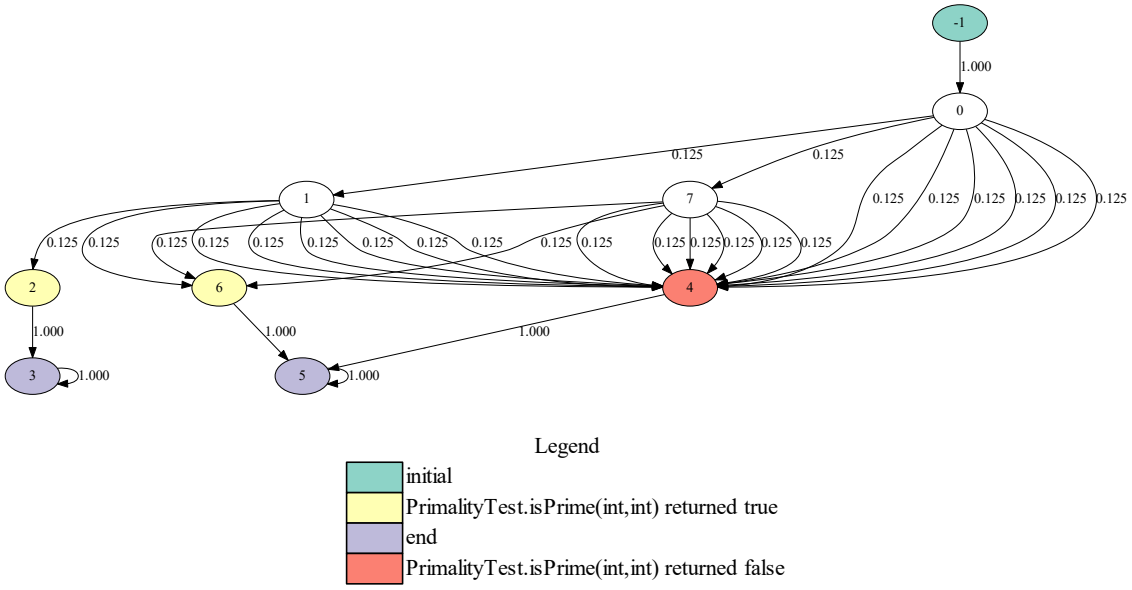


Figure 4.1: The state space for the Miller-Rabin primality test run for two iterations with the input number 9.

In states 2 and 6, the algorithm incorrectly identifies the composite number 9 as a prime. These states are labelled with `"true__MillerRabinPrimalityTest_isPrime__II__Z"`. This label, abbreviated below as `"incorrect"`, captures that the method

`isPrime` of the class `MillerRabinPrimalityTest`, which takes two arguments of type `int` and returns a value of type `boolean`, returns the value `true`.

The probability that the Miller-Rabin primality test returns the wrong result can be computed by using JPF and PRISM in tandem. We modify line 10 of the configuration file described above as follows.

```
10 listener = probabilistic.listener.StateSpaceText;
   ↪ label.StateLabelText
```

As a result, JPF creates the file `MillerRabinPrimalityTest.tra` that contains the transitions and their probabilities, and the file `MillerRabinPrimalityTest.lab` that contains the state labelling. Together they specify a labelled Markov chain. Subsequently, we use our converter to transform the labelled Markov chain into PRISM's format.

Finally, we use PRISM to compute for this labelled Markov chain 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 `MillerRabinPrimalityTest` returns `true`, is reached. PRISM returns the probability `0.0625`, which corresponds to reaching either state 2 or state 6 in Figure 4.1. Note that the probability that a composite number is erroneously reported to be prime is at most 2^{-2k} [Rab80], where $k = 2$.

By running the model checking tool with different command line arguments in line 2 of the configuration file, we obtain a collection of error probabilities that can be graphed as shown in Figure 4.2. As this example demonstrates, PRISM can provide quantitative information that enriches the qualitative verification results of JPF.

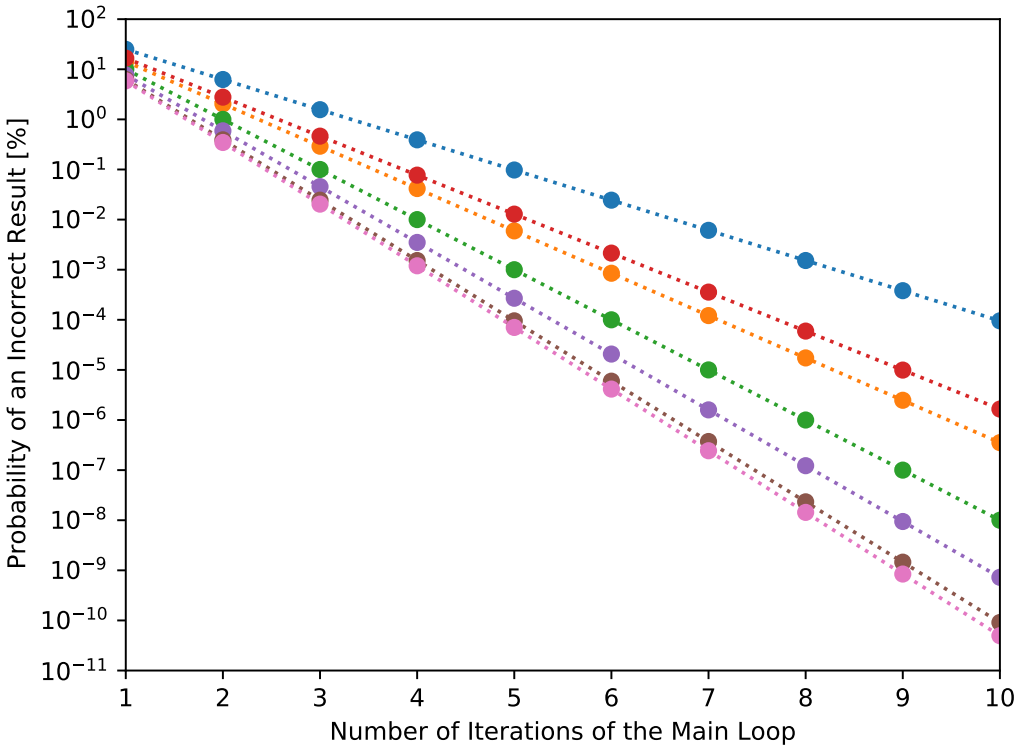


Figure 4.2: This graph depicts the results of the model checking tool applied to the Java code implementing the Miller-Rabin primality test. The colours represent the following different composite number inputs:
● = 9, ● = 15, ● = 21, ● = 25, ● = 27, ● = 33, ● = 35.

4.2 Quicksort

Quicksort is an efficient and commonly used sorting algorithm developed by Hoare [Hoa61]. Like most algorithms based on the divide-and-conquer paradigm, quicksort is recursive. The worst-case expected running time of the algorithm is $\mathcal{O}(n \log n)$; however, the absolute worst-case running time is $\mathcal{O}(n^2)$. This occurs when, at every recursive step of Algorithm 2, a pivot is chosen that partitions the array of size n into two, such that one sub-array is empty and the other sub-array has size $n - 1$. We compute the probability that such an execution takes place. We also compute the probability that a good-case occurs. We define a good case as when each of the two sub-arrays is of size at least $\frac{1}{4}n$ and at most $\frac{3}{4}n$, when $n > 2$.

Algorithm 2: QuickSort(S)

Input: $(S, <)$ a total order of n elements
Output: S sorted in increasing order
1 choose an element y uniformly at random from S
2 $S_1, S_2 \leftarrow \emptyset$
3 **foreach** $x \in S \setminus \{y\}$ **do**
4 **if** $x < y$ **then** $S_1 \cup \{x\}$ **else** $S_2 \cup \{x\}$
5 **end**
6 **return** QuickSort(S_1) + y + QuickSort(S_2)

We have implemented the algorithm in Java in a class named `QuickSort`. The randomization in line 1 is realized by jpf-probabilistic's `UniformChoice.make` method. Let us first consider the property corresponding to the worst-case. We need to add ghost variables to the Java implementation to capture whether one of the sub-arrays

is empty. Assume we add the following two fields to the class `QuickSort`.

```
/* true if S1 is empty */
private static boolean smallerIsEmpty;
/* true if S2 is empty */
private static boolean largerIsEmpty;
```

Between steps 5 and 6 of Algorithm 2, we set the boolean field `smallerIsEmpty` to `true` if $S_1 = \emptyset$ and `false` otherwise. Immediately afterwards, we set the boolean field `largerIsEmpty` to `true` if $S_2 = \emptyset$ and `false` otherwise.

We then create the following configuration file to label these fields and generate a graphical representation of the state space. We provide as an argument the array [8, 2, 4] to sort.

```
target = QuickSort
target.args = 8,2,4
classpath = <directory containing QuickSort.class>

@using jpf-label
label.class = label.BooleanStaticField
label.BooleanStaticField.field = QuickSort.smallerIsEmpty;
   ↪ QuickSort.largerIsEmpty

@using jpf-probabilistic
listener = probabilistic.listener.StateSpaceDot;
   ↪ probabilistic.listener.StateLabelVisitor
probabilistic.listener.StateSpaceDot.precision = 2
```

Running JPF with these application properties results in the labelled Markov chain depicted in Figure 4.3. States in which the field `smallerIsEmpty` is `true` are coloured purple, while states in which this field is `false` are coloured yellow. States in which the field `largerIsEmpty` is `true` are coloured red, while states in which this field is

`false` are coloured green. Thus, since we would like to compute the probability of the worst case, that is, consistently having one of the sub-arrays empty, we will compute the probability of taking a path, from state 0 to any of the final states 7, 8, or 15, along which every state is coloured purple or red. We do not consider paths starting from the initial state -1, since the fields have not yet been initialized in this state and as such have the default value of false.

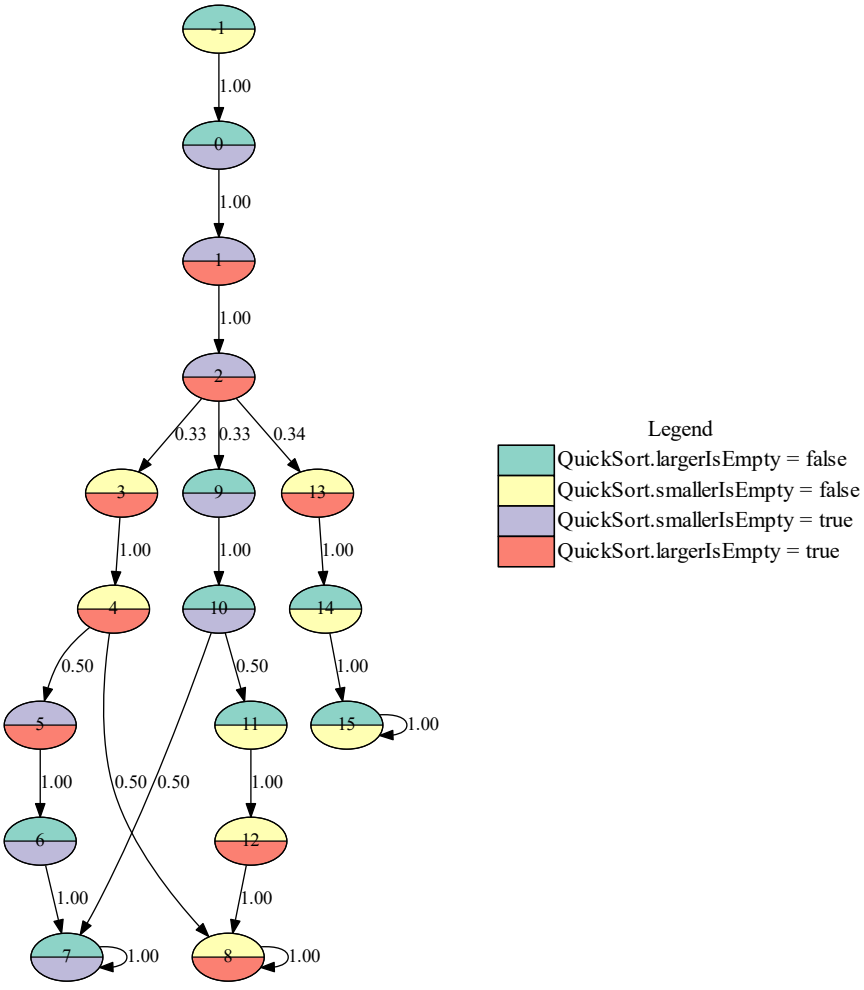


Figure 4.3: The state space for quicksort, with two ghost fields, for three elements.

Consider the case in which the element 2 is chosen as the pivot in line 1 of Algorithm 2. When we construct the sub-arrays in line 2, S_1 will be empty and S_2 will contain the rest of the elements, namely 8 and 4. Thus, at this point, we will set the boolean field `smallerIsEmpty` to `true` and then the boolean field `largerIsEmpty` to `false`. Then in line 3, S_1 is trivial and does not need to be recursively sorted, however S_2 will be sorted recursively. Assume that the element 8 is now chosen as the pivot. In this case, S_1 will consist of the element 4 and S_2 will now be empty. We now change the boolean field `smallerIsEmpty` to `false`. Since the transition is broken after every modification of a specified field, as described in Section 2.1.2, the transition will be broken after this assignment and a new state will be introduced in which both fields `smallerIsEmpty` and `largerIsEmpty` are labelled as `false`. We then set the boolean field `largerIsEmpty` to `true`. This execution satisfies the property described above, since at every recursive step one of the sub-arrays is empty. However, the intermediate state which falsely indicates that none of the sub-arrays is empty results in this execution not being included when computing the probability of the worst-case. This scenario corresponds to the path from state -1 to the final state 8 through the problematic state 11 in Figure 4.3. Recall that we are looking for execution paths along which every state is coloured purple or red.

As this example has shown, adding more than one ghost variable per prop-

erty can be tricky as the variables may interleave and lead to unexpected results. Therefore, we only add a single field which captures if either of the sets S_1 or S_2 is empty.

```
1 /* true if either S1 or S2 is empty */
2 private static boolean oneIsEmpty;
```

Between steps 5 and 6 of Algorithm 2, we set this boolean field `oneIsEmpty` to `true` if $S_1 = \emptyset \vee S_2 = \emptyset$ and `false` otherwise. Consequently, we modify line 7 of the configuration file as follows.

```
7 label.BooleanStaticField.field = QuickSort.oneIsEmpty
```

As shown in the resulting labelled Markov chain depicted in Figure 4.4, using fewer ghost variables gives rise to a simpler state space.

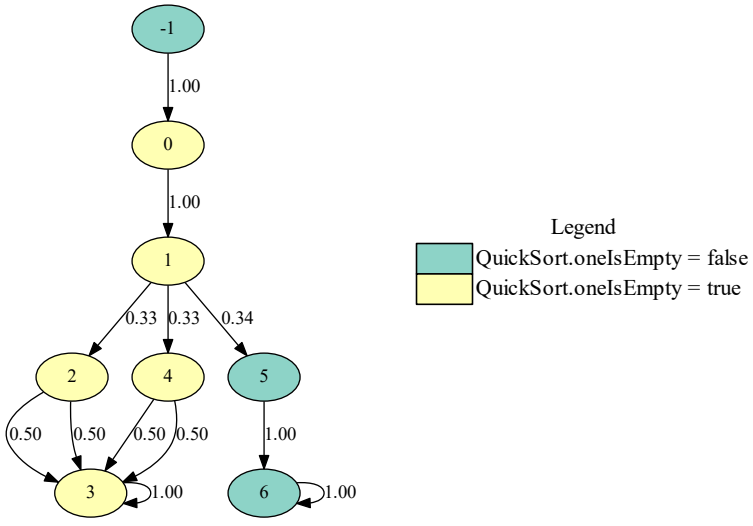


Figure 4.4: The state space for quicksort, with one ghost field, for three elements.

Let us now consider the property corresponding to a good case. We add the

following boolean field to capture if both sub-arrays are of size at least $\frac{1}{4}n$ and at most $\frac{3}{4}n$, when the array size, n, is greater than 2.

```
1 /* true if n≤2 or each sub-array's size is ≥1/4n and ≤3/4n */
2 private static boolean sizesWithinRange;
```

Between steps 5 and 6 of Algorithm 2, after setting the boolean field `oneIsEmpty`, we set the boolean field `sizesWithinRange` to `true` if $n \leq 2 \vee \frac{1}{3} \leq \frac{S_1.size}{S_2.size} \leq 3$ and `false` otherwise. Note that this captures the requirement for a good case described above.

We modify line 7 of the configuration file to label the new boolean field as well.

```
7 label.BooleanStaticField.field = QuickSort.oneIsEmpty;
    ↪ QuickSort.sizesWithinRange
```

We also modify line 10 to specify that JPF should provide a textual representation of the labelled Markov chain.

```
10 listener = probabilistic.listener.StateSpaceText;
    ↪ label.StateLabelText
```

Subsequently, we run JPF with these application properties and then run our converter `JPFtoPRISM` to transform the resulting labelled Markov chain into PRISM's format.

Finally, we use PRISM to determine the probability of the worst-case occurring by computing the property `P=? [ X G "true__QuickSort_oneIsEmpty" ]`. The property specifies that from the next state after the initial state, all states along

the path are labelled with `"true__QuickSort_oneIsEmpty"`. We do not consider the initial state, as mentioned above, since the boolean field `oneIsEmpty` has not yet been initialized in this state. PRISM returns the result `0.66666`, which corresponds to the probability of a path from state 0 to state 3 being taken in Figure 4.4.

Additionally, to determine the probability of a good case taking place, we compute the property `P=? [ X X G "true__QuickSort_sizesWithinRange" ]`. That is, PRISM computes the probability of all paths, beginning two states after the initial state, in which all states are labelled with `"true__QuickSort_sizesWithinRange"`. We disregard the initial state for the reason described above. We also disregard the state after the initial state, since the transition will be broken after the initialization of the first boolean field `oneIsEmpty` and in the resulting state the field `sizesWithinRange` will not yet be initialized and still have the default value of `false`. Thus, the boolean field `sizesWithinRange` is only initialized in the third state. For this property, PRISM returns the probability `0.33334`.

When boolean fields are declared, they take the default value `false` and only after that they are initialized. So if a field is initialized to `true`, in the initial state it will have the value `false` and then in a later state, once it is initialized, it will have the value `true`. As a result, if we are interested in paths in which particular fields are `true`, we need to exclude at least the initial state. In order to avoid this, especially with more complex models, one could negate the property

of interest. If the field is initialized to `true`, negate it so that it is initialized to `false`. Therefore, the initial state(s) need not be excluded. For example, instead of defining the field `oneIsEmpty`, we create the opposite field `noneAreEmpty`. We set this field to `true` if $S_1 \neq \emptyset \wedge S_2 \neq \emptyset$ and `false` otherwise. The resulting labelled Markov chain is depicted in Figure 4.5. We then negate the property `P=? [ X G "true__QuickSort_oneIsEmpty" ]` by computing the property `P=? [ G "false__QuickSort_noneAreEmpty" ]` instead. Notice that we no longer use the next operator `X`, since we do not need to exclude any states from the path.

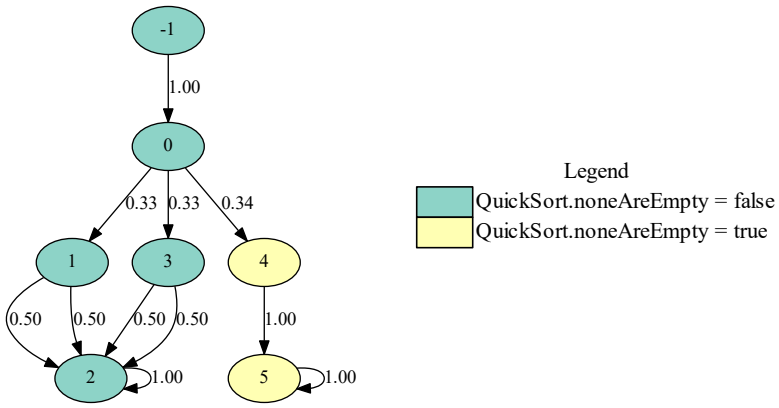


Figure 4.5: The state space for quicksort when using the negated ghost field, for three elements.

By varying the size of the array in line 2 of the configuration file and computing the probabilities of the two properties discussed above, we obtain the graph shown in Figure 4.6. The probability of the worst-case occurring decreases as the size of the input array increases. Similarly, the probability of a good case taking place

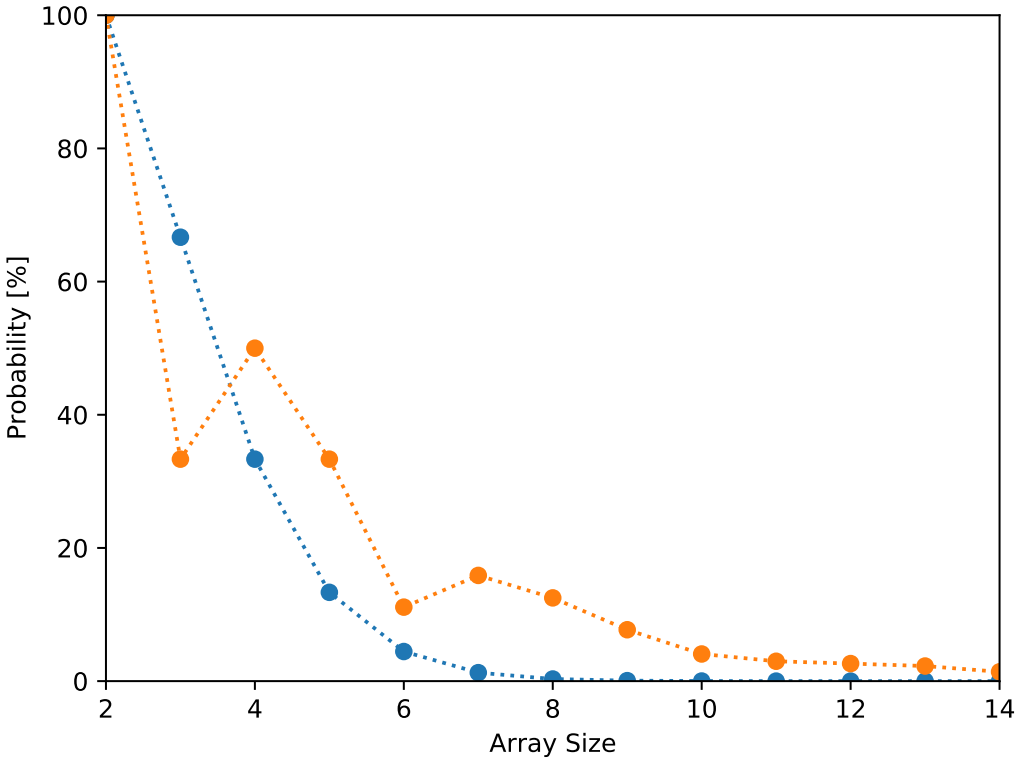


Figure 4.6: This graph depicts the results of the model checking tool applied to the Java code implementing the Quicksort algorithm. The colours represent the properties: ● = worst-case, ● = good case

generally decreases as the size of the input array increases, with a few exceptions. Recall that we define a good case as when each of the two sub-arrays is of size at least $\frac{1}{4}n$ and at most $\frac{3}{4}n$, where $n > 2$ is the size of the input array. For example, when the size of the input array is three, a good case can only occur if the middle element is picked as the pivot, as both sub-arrays will be of size one. If either the largest element or the smallest element in the array is chosen as the pivot, one of the sub-arrays will be empty. Thus, when the size of the input array is three, a

good case occurs with probability $\frac{1}{3} \approx 0.33$. On the other hand, when the size of the input array is four, if either of the middle two elements is picked as the pivot, one of the sub-arrays will be of size one and the other sub-array will be of size two, resulting in a good case. If either the largest element or the smallest element in the array is chosen as the pivot, one of the sub-arrays will be empty. Thus, when the size of the input array is four, a good case occurs with probability $\frac{2}{4} = 0.5$. Therefore, there is an increase in the probability that a good case occurs between the array sizes three and four in Figure 4.6. The increase in the probability that a good case occurs between array sizes six and seven can be explained similarly.

4.3 Lazy Select

Let us consider the lazy select algorithm defined in Algorithm 3, a variation of an algorithm due to Floyd and Rivest [FR75]. This Las Vegas 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, [XT15]). Step 12 of the algorithm may fail with a small probability. If that happens, the algorithm is called recursively and steps 1–12 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 detect 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.

Algorithm 3: LazySelect(S, k)

Input: $S \subseteq \mathbb{Z}$ a set of n elements
$k \in \mathbb{N}$ an integer in $[1, n]$
Output: $S_{(k)}$, the k^{th} smallest element of S

build a multiset R by choosing $n^{3/4}$ elements, independently and uniformly at random with replacement, from S
sort R
$x \leftarrow kn^{-1/4}$
$l \leftarrow \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$
$h \leftarrow \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$
$a \leftarrow R_{(l)}$
$b \leftarrow R_{(h)}$
determine the ranks of a and b, $r_S(a)$ and $r_S(b)$, by comparing a and b to every element of S
if $k < \lceil n^{1/4} \rceil$ **then** $P \leftarrow \{y \in S \mid y \leq b\}$
else if $k > \lfloor n - n^{1/4} \rfloor$ **then** $P \leftarrow \{y \in S \mid y \geq a\}$
else $P \leftarrow \{y \in S \mid a \leq y \leq b\}$
if $S_{(k)} \in P \wedge |P| \leq 4n^{3/4} + 2$ **then**
    sort P
    if $k < \lceil n^{1/4} \rceil$ **then return** $P_{(k)}$ **else return** $P_{(k - r_S(a) + 1)}$
else
    return LazySelect(S, k)
end

Let $r_S(t)$ denote the rank of an element t in a set S (the k^{th} smallest element has rank k) and let $S_{(i)}$ denote the i^{th} smallest element of S. We extend the use of this notation to subsets of S as well. Thus, we seek to identify $S_{(k)}$ in Algorithm 3.

We implemented the algorithm in Java in a class called `LazySelect`. We use jpf-probabilistic's `UniformChoice.make` method to capture the randomization in line 1. Assume we want to find the third smallest number in the array [9, 7, 1, 5, 6]. To achieve this, we create the following application properties file.

```
1  target = LazySelect
2  target.args = 3,9,7,1,5,6
3  classpath = <directory containing LazySelect.class>
4
5  @using jpf-label
6  label.class = label.Initial; label.End
7
8  @using jpf-probabilistic
9
10 search.class = gov.nasa.jpf.search.heuristic.BFSHeuristic
11 listener = probabilistic.listener.StateSpaceText;
      ↪ label.StateLabelText;gov.nasa.jpf.listener.BudgetChecker
12 budget.max_heap=10240M
```

In line 2, we provide the command line arguments mentioned above. Line 6 specifies that the initial state and the final states should be labelled. In line 10 we specify that JPF should use its breadth first search strategy to explore the state space. Line 11 specifies that JPF should generate a textual representation of the state space and we also specify that JPF should use the budget checker since the state space is infinite. Line 12 captures that the budget for the heap should be a maximum of 10 GB.

When we use JPF, extended with jpf-label and jpf-probabilistic, to model check the algorithm with this configuration file, JPF runs out of its 10 GB of memory

after 2 minutes and 13 seconds. In that time, JPF visits 1,161,233 states and traverses 1,713,457 transitions 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. To demonstrate how we do this, let us consider the first 20 states of the example. In order to specify that JPF should generate a textual and graphical representation of a limited part of the labelled state space in which the explored states are marked, we modify lines 11 and 12 of the configuration file as shown below. We also specify that the probabilities of the transitions should be depicted with one digit precision in the graphical representation of the labelled state space.

```
11 listener = probabilistic.listener.StateSpaceDot;
     ↪ probabilistic.listener.StateLabelVisitor;
     ↪ probabilistic.listener.ExploredStatesVisitor;
     ↪ probabilistic.listener.StateSpaceText;
     ↪ label.StateLabelText;gov.nasa.jpf.listener.BudgetChecker
12 budget.max_state=20
13 probabilistic.listener.StateSpaceDot.precision = 1
```

The resulting graphical representation of the state space is depicted in Figure 4.7. Only the initial state -1 is labelled since a final state has not been reached yet. States 4–21 have not been fully explored by JPF, as indicated by the single circle

around those states.

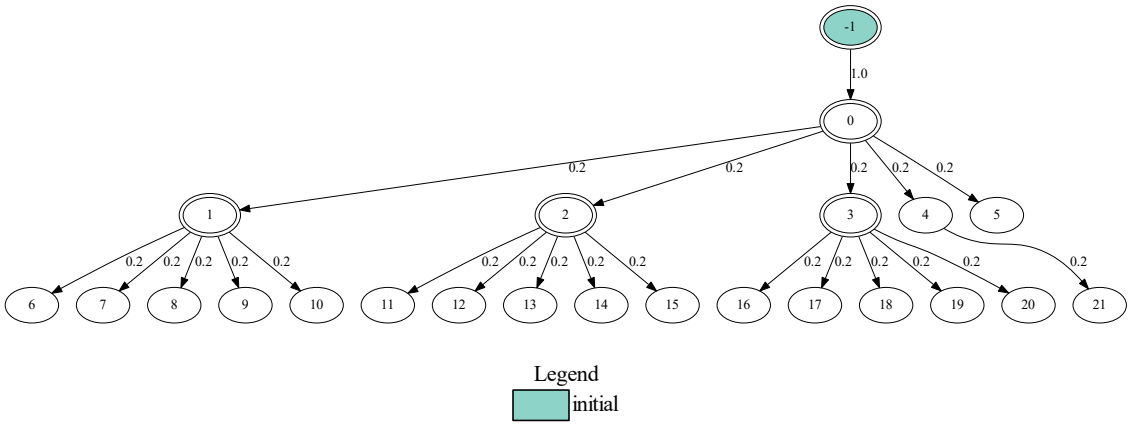


Figure 4.7: The first twenty states of the state space for the lazy select algorithm for finding the third smallest of five elements.

We run our converter `JPFtoPRISM` which transforms the textual representation of the labelled Markov chain into the PRISM format. Whenever JPF has not fully explored the state space, the converter also adds a labelled sink state and a transition to this sink state from all states that have not yet been fully explored by JPF. Thus, the converter will add a new state 22, labelled with `"sink"` with a transition from state 4 to state 22 with probability `0.8` and transitions from states 5–22 to state 22 with probability `1.0` each. In this way, the converter completes the labelled Markov chain, allowing us to use PRISM to check properties of the Java code.

Similarly, we run the converter on the above mentioned labelled Markov chain with 1,161,233 states, to complete the state space. Finally, we can use PRISM to

determine the probability that the sink state is eventually reached by computing the property `P=? [ F "sink" ]`. PRISM returns a value less than 0.000035. As a consequence, with more than probability 0.999965 only fully explored states are reached. Hence, if we run the Java code then with at least probability 0.999965 we will not encounter any violation of the properties checked by JPF. This number represents the progress made by JPF [ZvB11].

By default, JPF uses depth-first search to traverse the state space. It also supports breadth-first search. Since jpf-probabilitic associates probabilities with the transitions, these probabilities can be used to drive the search of the state space. The extension provides four such search strategies. Probability-first search (PFS), which was introduced by Zhang in [Zha10], 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) [Zha10] 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 [Tan13], Tang introduced two search strategies inspired by reinforcement learning [SB18]. 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. An earlier version of jpf-probabilistic has been discussed in [ZvB10]. Since then, the search strategies SMS and EGS have been added and the search strategies PFS and RS have been implemented more efficiently.

To compare the progress made over time for these search strategies, we vary the search strategy used by JPF in line 10 and limit the search by time in line 12 of the configuration file in increments of 50ms. We use EGS with $\epsilon = 0.1$ and SMS with $\tau = 0.5$. Since different search strategies may visit states in different

orders, they may make progress at different rates. As shown in Figure 4.8, this is indeed the case for the Java implementation of lazy select. Except for SMS, the search strategies that take the probabilities into account make more progress than breadth-first search. Depth-first search, JPF's default search strategy, makes no progress for this particular example. The graph for depth-first search coincides with the x-axis in Figure 4.8.

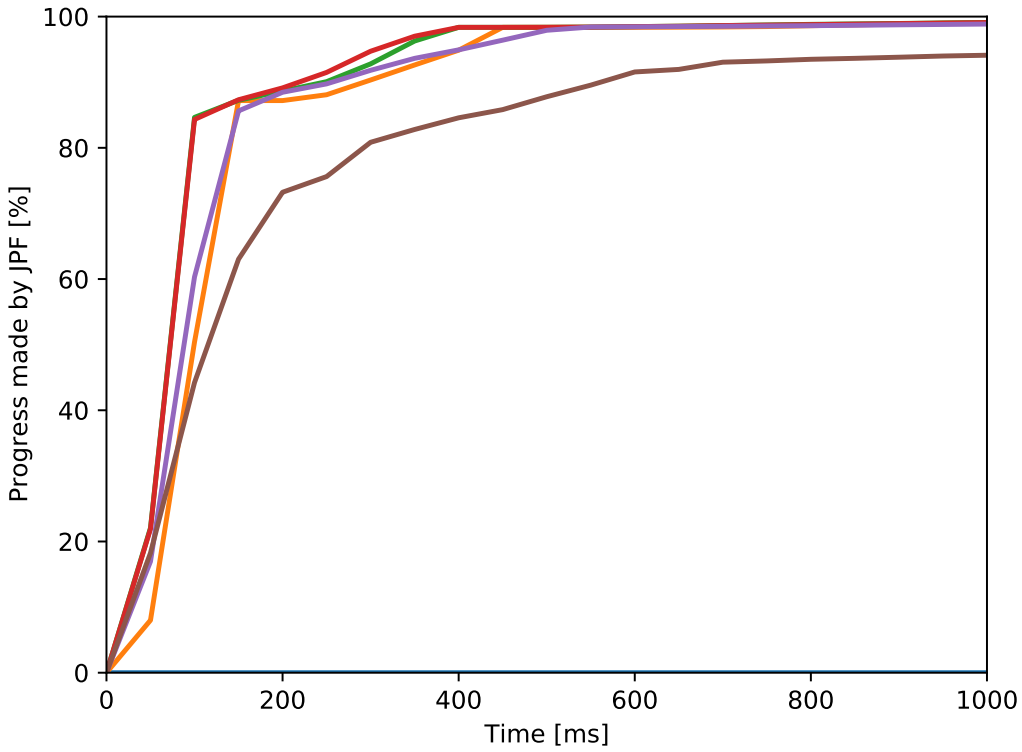


Figure 4.8: This graph depicts the results of the model checking tool applied to the Java code implementing lazy select that selects the third smallest of five elements. The colours represent the different search strategies:
● = depth-first search, ● = breadth-first search, ● = ϵ-greedy search,
● = probability-first search, ● = random search, ● = softmax search.

4.4 Unbiased Toss of a Biased Coin

Von Neumann proposed that we can reconstruct a 50-50 chance out of a biased coin by making two independent tosses [vN51]. If we get heads-heads or tails-tails, we reject the tosses and try again. If we get heads-tails or tails-heads, we accept the result as heads or tails, respectively. Let `0` represent heads and `1` represent tails, then this process is formalized in Algorithm 4. To confirm that the algorithm simulates a fair coin, we will compute the probability of the algorithm returning heads and tails.

Algorithm 4: Toss(p)

Input: $p \in \mathbb{R}$ a floating-point number in $(0, 1)$
Output: 0 for heads or 1 for tails

1 $x \leftarrow 0$ with probability p and 1 with probability $p - 1$
2 $y \leftarrow 0$ with probability p and 1 with probability $p - 1$
3 **if** $(x = 0 \wedge y = 0) \vee (x = 1 \wedge y = 1)$ **then return** Toss(p) **else return** x

We implemented the algorithm in Java in a method called `flip` of a class named `FairBiasedCoin`. The randomized choices in lines 1–2 are captured by jpf-probabilistic's `Choice.make` method with the array $[p, 1-p]$ passed as the argument. We will label the return of the integer method `flip`, so that we can determine the probability that the method returns heads and the probability that it returns tails. We create the following application properties file.

```
1 target = FairBiasedCoin
2 target.args = 0.7
3 classpath = <directory containing FairBiasedCoin.class>
4
```

```
5  @using jpf-label
6  label.class = label.Initial; label.End;
      ↪ label.ReturnedIntegerMethod
7  label.ReturnedIntegerMethod.method =
      ↪ FairBiasedCoin.flip(double)
8
9  @using jpf-probabilistic
10 listener = probabilistic.listener.StateSpaceDot;
      ↪ probabilistic.listener.StateLabelVisitor
11 probabilistic.listener.StateSpaceDot.precision = 1
```

Line 1 specifies that the Java app named `FairBiasedCoin` is to be model checked by JPF. In line 2, we provide the command line arguments, namely `0.7` which is the bias of the coin. Line 6 specifies that the initial state and the final states should be labelled, as well as those states in which the integer method `flip` returns, as specified by the method signature in line 7. Finally, line 10 specifies that JPF should generate a labelled graphical representation of the state space and line 11 captures that the probabilities of the transitions should be depicted with one digit precision. Running JPF with this configuration file results in the creation of the file named `FairBiasedCoin.dot`, containing the labelled Markov chain depicted in Figure 4.9.

The initial state -1 and the final state 5 are labelled, as well as those states in which the integer method that reports the result of the coin flip returns, that is, states 4 and 7. State 4 is labelled with `"0__FairBiasedCoin_flip__D__I"`, which captures that the method `flip` of the class `FairBiasedCoin`, which takes an argu-

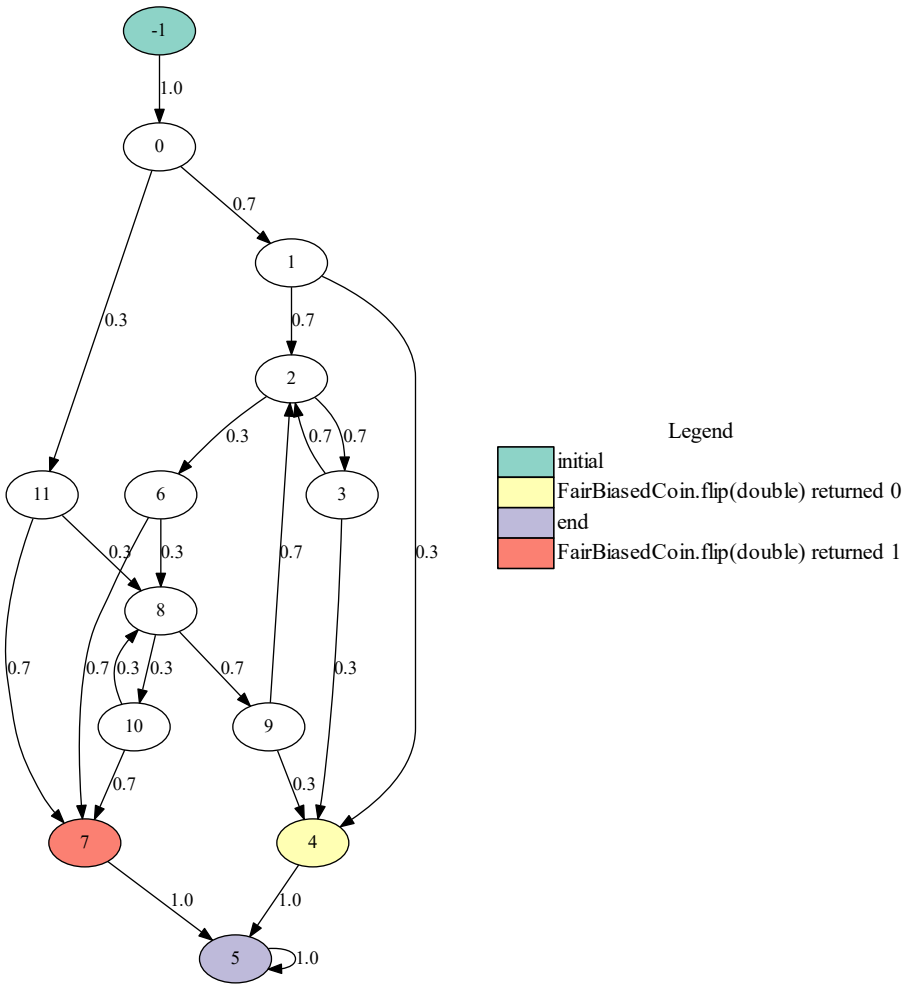


Figure 4.9: The state space for the fair biased coin toss algorithm run with the input bias 0.7.

ment of type `double` and returns a value of type `int`, returns `0` which represents heads. Similarly, state 7 is labelled with `"1__FairBiasedCoin_flip__D__I"`, which captures that the above mentioned method returns the value `1` which represents tails.

The probability that the biased coin toss algorithm returns heads and tails can

be computed by using both JPF and PRISM. We modify line 10 of the configuration file described above as follows.

```
10 listener = probabilistic.listener.StateSpaceText;
   ↪ label.StateLabelText
```

As a result, JPF generates a transition file and a label file, which specify the labelled Markov chain shown in Figure 4.9. Subsequently, we use our converter to transform the labelled Markov chain into PRISM's format.

Finally, we use PRISM to compute for this labelled Markov chain the property `P=? [ F "0__FairBiasedCoin_flip__D__I" ]` to determine the probability with which the algorithm returns heads. PRISM returns the probability `0.499999`, which corresponds to reaching state 4 in Figure 4.9. Similarly, we check the property `P=? [ F "1__FairBiasedCoin_flip__D__I" ]` to obtain the probability with which the algorithm returns tails. PRISM also returns the probability `0.499999`, which corresponds to reaching state 7 in Figure 4.9.

By running the model checking tool with different command line arguments in line 2 of the configuration file and computing the probability of getting heads, we construct the graph illustrated in Figure 4.10. Note that the probabilities of getting tails are the same as the probabilities for getting heads. If PRISM were to use exact real arithmetic, it would return exactly 0.5 for every possible value of bias.

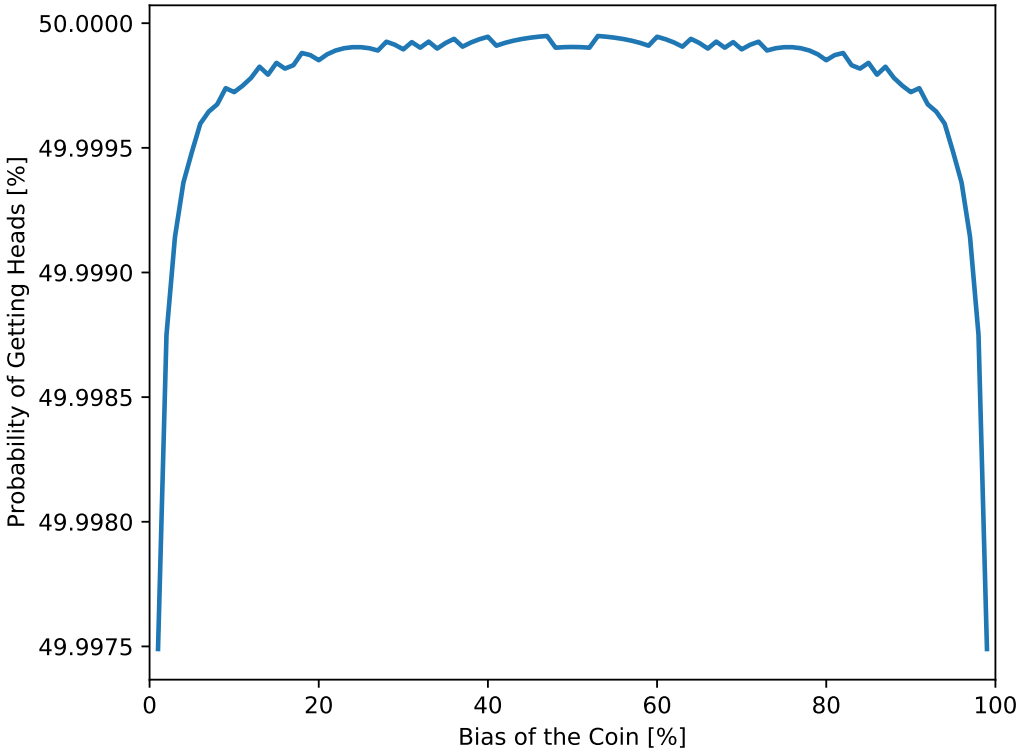


Figure 4.10: This graph depicts the results of the model checking tool applied to the Java code implementing the algorithm to simulate a fair coin flip with a biased coin.

4.5 Randomized Binary Search

Consider the binary search algorithm, which finds the index of a given target value within a sorted array. The main idea of the algorithm is to keep track of the part of the array in which the target value could possible be, and pick the middle element from that range as the pivot. The binary search algorithm can be randomized by choosing a random element from the current range, instead of the middle element, as shown in Algorithm 5. Using our tool, we can determine the probability that

the randomized algorithm does worse than the deterministic algorithm.

Algorithm 5: Search(A, t)

Input: $A \subseteq \mathbb{Z}$ an array of n sorted elements
$t \in \mathbb{Z}$ an integer
Output: The index of t in A if present, -1 otherwise

```
1 left ← 0
2 right ← n − 1
3 while left ≤ right do
4     choose pivot uniformly at random from [left, right]
5     if t = A[pivot] then return pivot
6     if t < A[pivot] then right ← pivot − 1 else left ← pivot + 1
7 end
8 return −1
```

We implemented the randomized binary search algorithm in Java in a class named `RandomizedBinarySearch`. The random choice in line 4 is captured by jpf-probabilistic's `UniformChoice.make` method. We added a boolean ghost field called `isWorse`. The field is true when the randomized algorithm takes more iterations of the while loop than the deterministic algorithm and false otherwise.

We specify that jpf-label should label the states of the model with the value of the boolean field `isWorse`, using the following application properties file.

```
1 target = RandomizedBinarySearch
2 target.args = 2,1,2,3,4,5
3 classpath = <directory containing
    ↪ RandomizedBinarySearch.class>
4
5 @using jpf-label
6 label.class = label.BooleanStaticField
7 label.BooleanStaticField.field =
    ↪ RandomizedBinarySearch.isWorse
8
9 @using jpf-probabilistic
```

```
10 listener = probabilistic.listener.StateSpaceDot;
   ↪ probabilistic.listener.StateLabelVisitor
```

Line 1 specifies that the Java app named `RandomizedBinarySearch` is to be model checked by JPF. In line 2, we provide the command line arguments, namely `2`, which is the target value, and the sorted array `[1,2,3,4,5]`. Line 6 specifies that the states should be labelled with the value of the boolean field `isWorse`, as specified by the field signature in line 7. Finally, line 10 specifies that JPF should generate a labelled graphical representation of the state space, with the default of two digits precision for the probabilities of the transitions. Running JPF with this configuration file results in the labelled Markov chain depicted in Figure 4.11.

The states coloured yellow are those states in which the randomized algorithm does worse than the deterministic algorithm. Thus, these states are labelled with `"true__RandomizedBinarySearch_isWorse"`, which captures that the field `isWorse` of the class `RandomizedBinarySearch` is `true`.

The probability that the randomized algorithm performs worse than the deterministic algorithm can be computed by using both JPF and PRISM. We modify line 10 of the configuration file described above as follows.

```
10 listener = probabilistic.listener.StateSpaceText;
   ↪ label.StateLabelText
```

As a result, JPF generates a transition file and a label file, which specify the labelled Markov chain shown in Figure 4.11. Subsequently, we use our converter to

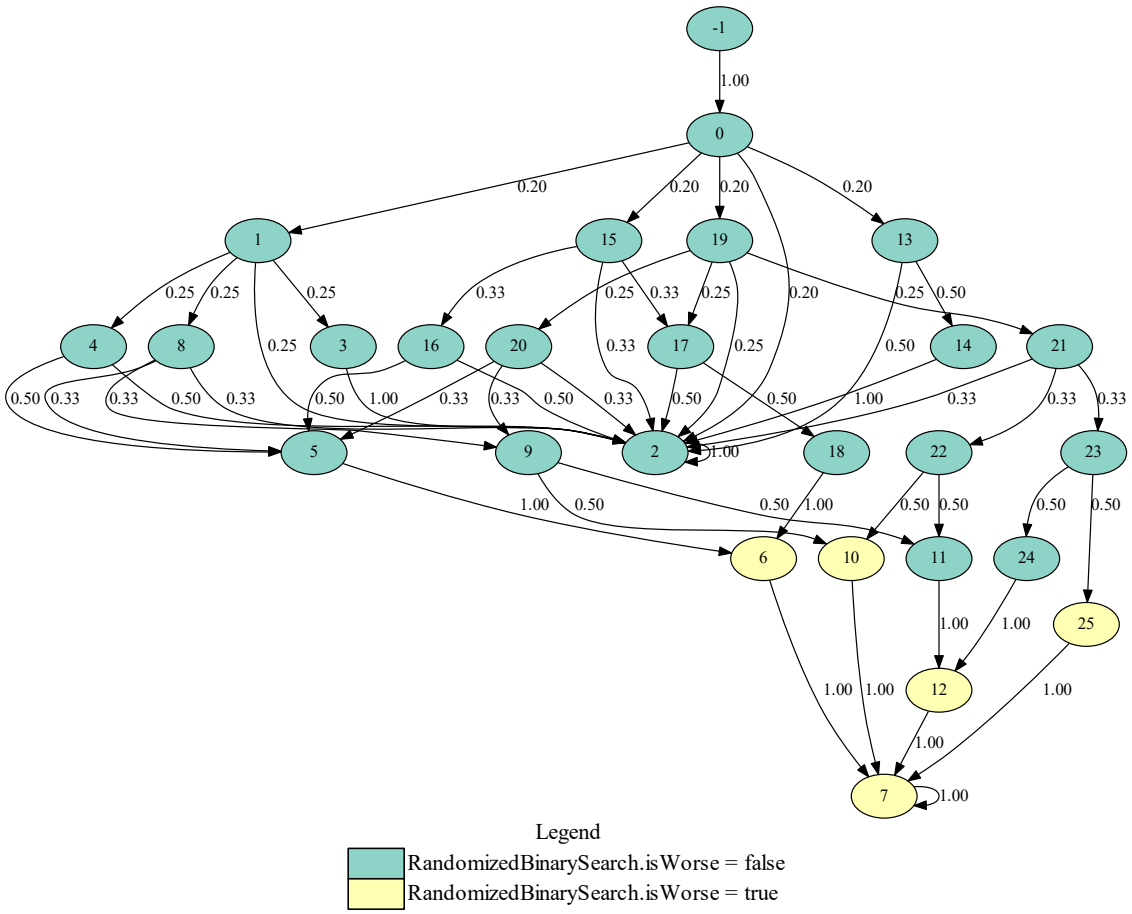


Figure 4.11: The state space for the randomized binary search algorithm run with the input array $[1, 2, 3, 4, 5]$ and target value 2.

transform the labelled Markov chain into PRISM's format.

Finally, we use PRISM to compute for this labelled Markov chain the property `P=? [ F "true__RandomizedBinarySearch_isWorse" ]` to determine the probability that we eventually reach a state in which the variable `isWorse` is true. PRISM measures that with a probability of `0.216667` the randomized algorithm requires more iterations than the deterministic algorithm, which corresponds to the probability

of reaching a yellow state in Figure 4.11.

By running the model checking tool with different command line arguments in line 2 of the configuration file and computing the probability that the randomized algorithm performs worse, we construct the graph illustrated in Figure 4.12.

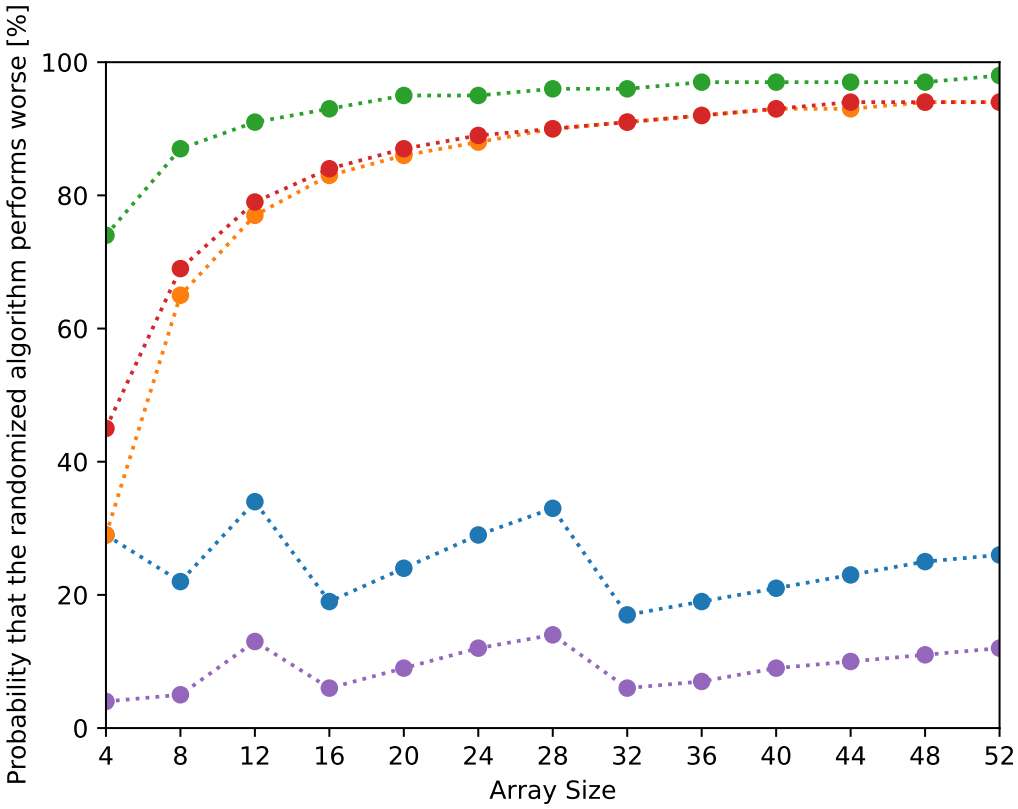


Figure 4.12: This graph depicts the results of the model checking tool applied to the Java code implementing the randomized binary search algorithm. The colours represent the following positions for the target values:
● $= 1$, ● $= n/4$, ● $= n/2$, ● $= 3n/4$, ● $= n$.

The dips in the graph correspond to the number of iterations taken by the deterministic algorithm. For instance, the deterministic algorithm always finds the

value at position $\frac{n}{2}$ in a single iteration and the value at position $\frac{n}{4}$ or $\frac{3n}{4}$ in two iterations, so the curves corresponding to these positions are smooth. However, when the target value is at position n, that is, the last element of the array, the deterministic algorithm takes three iterations when the size of the array $n = 4$, four iterations when $n \in \{8, 12\}$, five iterations when $n \in \{16, 20, 24, 28\}$, and so on. Similarly, the dips in the curve when the target value is the first element of the array correspond to the an increase in the number of iterations taken by the deterministic algorithm.

Note that when the target value is not present in the input array, the number of iterations taken by the algorithm depends on the target value. For example, if the target value is larger than all of the elements in the array, the probability that the randomized algorithm performs worse than the deterministic algorithm would be similar to when we search for the last element in the array. Likewise, if the target value is smaller than all of the elements in the array, the probability would be similar to when we search for the first element. Generally, the number of iterations is influenced by the position where the target value would be placed in the sorted array.

4.6 Summary

PRISM can be used to augment JPF's qualitative results with quantitative information. 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.

We have applied our tool to sixty randomized algorithms including those presented in this chapter. The Java implementations of these algorithms accompany jpf-probabilistic, from which we can generate a large collection of practical labelled Markov chains. For all of these examples, the overhead of jpf-label and jpf-probabilistic is very limited.

5 Probabilistic Bisimilarity

In this chapter, we review definitions that we will use in the coming chapters. Most of this material can be found in [vB17]. In Chapters 3 and 4 we have already seen numerous examples of labelled Markov chains. In this chapter we formalize this notion. Furthermore, we introduce the ideas of an equivalence relation and a partition refinement algorithm. Finally, we present the concept of probabilistic bisimilarity.

5.1 Labelled Markov Chain

Given a nonempty and finite set S, we denote the set of probability distributions on S by $\mathcal{D}(S)$. Recall that probability distribution on S is a function $\delta : S \to [0, 1]$ such that $\sum_{s \in S} \delta(s) = 1$.

Definition 1. *A labelled Markov chain is a tuple $\langle S, L, \tau, \ell \rangle$ consisting of*

- *a nonempty and finite set S of states,*

- *a nonempty and finite set L of labels,*
- *a transition function $\tau : S \to \mathcal{D}(S)$, and*
- *a labelling function $\ell : S \to 2^L$.*

For example, consider the labelled Markov chain depicted in Figure 5.1. The set of states S is $\{s_0, s_1, s_2, s_3, s_4\}$, while the set of labels L is $\{\mathit{green}, \mathit{red}\}$. The labelling function ℓ defines the labels of each state, namely $\ell(s_0) = \{\mathit{green}\}$, $\ell(s_1) = \ell(s_2) = \emptyset$, and $\ell(s_3) = \ell(s_4) = \{\mathit{red}\}$. The transition function τ captures all of the transitions and their probabilities. For instance, the probability of transitioning from state s_1 to state s_4 is 0.3, thus $\tau(s_1)(s_4) = 0.3$.

5.2 Partition and Refinement

Recall that a relation $\mathcal{R} \subseteq S \times S$ is an *equivalence* relation if for all s, t, $u \in S$,

- $(s, s) \in \mathcal{R}$,
- if $(s, t) \in \mathcal{R}$ then $(t, s) \in \mathcal{R}$, and
- if $(s, t) \in \mathcal{R}$ and $(t, u) \in \mathcal{R}$ then $(s, u) \in \mathcal{R}$.

We denote the set of equivalence relations on S by $\mathcal{E}(S)$. Note that $S \times S$ is the largest equivalence relation on S. The set $\mathcal{E}(S)$ together with the relation $\subseteq$ forms

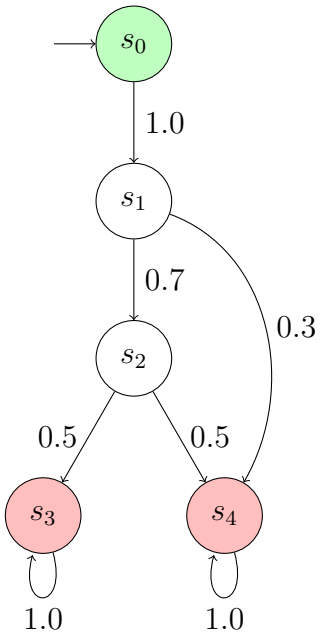


Figure 5.1: A labelled Markov chain with five states and five transitions where the initial and final states are labelled with green and red, respectively.

a partial order [DP02, Example 2.34]. We will exploit this structure later in this section.

Definition 2. *Let $\mathcal{R} \in \mathcal{E}(S)$. The set $S/\mathcal{R}$ is defined by*

$$S/\mathcal{R} = \{\, \{\, t \in S \mid (s,t) \in \mathcal{R} \,\} \mid s \in S \,\}.$$

The elements of $S/\mathcal{R}$ are known as the *$\mathcal{R}$-equivalence classes*. We will also call them *blocks*.

Proposition 1. *For all $\mathcal{R} \in \mathcal{E}(S)$,*

(a) $\bigcup S/\mathcal{R} = S$ and

(b) for all C, $D \in S/\mathcal{R}$, if $C \neq D$ then $C \cap D = \emptyset$.

Proof. Let $\mathcal{R} \in \mathcal{E}(S)$.

(a) Obviously, $\bigcup S/\mathcal{R} \subseteq S$. Since $\mathcal{R}$ is an equivalence relation, $(s, s) \in \mathcal{R}$ for all $s \in S$ and, hence, we can conclude that $S \subseteq \bigcup S/\mathcal{R}$.

(b) Assume that $C = \{\, t \in S \mid (s, t) \in \mathcal{R} \,\}$ and $D = \{\, u \in S \mid (u, v) \in \mathcal{R} \,\}$ and $C \neq D$. Towards a contradiction, assume that $C \cap D \neq \emptyset$, that is, $w \in C \cap D$. Then $(s, w) \in \mathcal{R}$ and $(u, w) \in \mathcal{R}$ and, therefore, $(s, u) \in \mathcal{R}$ since $\mathcal{R}$ is an equivalence relation. Next, we show that this implies $C = D$, which contradicts our assumption that $C \neq D$. Let $t \in C$. Then $(s, t) \in \mathcal{R}$. Since $(s, u) \in \mathcal{R}$, we have that $(u, t) \in \mathcal{R}$ since $\mathcal{R}$ is an equivalence relation. Hence, $t \in D$. Therefore, $C \subseteq D$. The opposite inclusion can be proved similarly.

□

Hence, $S/\mathcal{R}$ forms a *partition* of S.

Proposition 2. *For all $\mathcal{R}$, $\mathcal{S} \in \mathcal{E}(S)$ and $C \in S/\mathcal{S}$, if $\mathcal{R} \subseteq \mathcal{S}$ then $C = \bigcup\{\, \{\, t \in S \mid (s, t) \in \mathcal{R} \,\} \mid s \in C \,\}$.*

Proof. Let $\mathcal{R}$, $\mathcal{S} \in \mathcal{E}(S)$ with $\mathcal{R} \subseteq \mathcal{S}$ and $C \in S/\mathcal{S}$. Since $\mathcal{R}$ is an equivalence relation, $(s, s) \in \mathcal{R}$ for each $s \in C$. Hence, $C \subseteq \bigcup\{\, \{\, t \in S \mid (s, t) \in \mathcal{R} \,\} \mid s \in C \,\}$. To prove the opposite inclusion, let $s \in C$ and $(s, t) \in \mathcal{R}$. Because $\mathcal{R} \subseteq \mathcal{S}$ by

assumption, we have that $(s,t) \in \mathcal{S}$. Since C is an $\mathcal{S}$-equivalence class, we can conclude that $t \in C$. □

From the above proposition we can conclude that each $\mathcal{S}$-equivalence class C is the disjoint union $D_0 \cup \cdots \cup D_n$ of $\mathcal{R}$-equivalence classes $D_0, \ldots, D_n$. Hence, $\mathcal{R}$ is called a *refinement* of $\mathcal{S}$. The basic idea of a *partition refinement algorithm* is the following. Start with the initial partition consisting of a single equivalence class and repeatedly refine this partition until no further refinement is possible.

Let us make this a little more precise. Consider the function $\Phi : \mathcal{E}(S) \to \mathcal{E}(S)$ that will be used to refine the partition. We assume that Φ is *monotone*, that is, if $\mathcal{R} \subseteq \mathcal{S}$ then $\Phi(\mathcal{R}) \subseteq \Phi(\mathcal{S})$. Then the partition refinement algorithm amounts to the following.

```
1 R = S × S
2 while Φ(R) ⊂ R
3     R = Φ(R)
```

We claim that the fact that $\mathcal{R}$ is a *pre-fixed point* of Φ, that is, $\Phi(\mathcal{R}) \subseteq \mathcal{R}$ is a loop invariant. We annotate the above algorithm as follows.

```
1 R = S × S
2 {Φ(R) ⊆ R}
3 while Φ(R) ⊂ R
4     {Φ(R) ⊆ R}
5     R = Φ(R)
6     {Φ(R) ⊆ R}
7 {Φ(R) = R}
```

Initially, $\mathcal{R} = S \times S$ and, hence, the loop invariant $\Phi(\mathcal{R}) \subseteq \mathcal{R}$ holds. In

order to conclude that the loop invariant is maintained, it suffices to show that $\Phi(\mathcal{R}) \subseteq \mathcal{R}$ implies $\Phi(\Phi(\mathcal{R})) \subseteq \Phi(\mathcal{R})$. This immediately follows from the fact that Φ is monotone. If the algorithm terminates, then upon termination we have that $\Phi(\mathcal{R}) \subseteq \mathcal{R}$ and $\Phi(\mathcal{R}) \not\subset \mathcal{R}$ and, hence, $\Phi(\mathcal{R}) = \mathcal{R}$, that is, $\mathcal{R}$ is a *fixed point* of Φ.

To conclude that the loop terminates, assume that $|S| = n$. Initially, $|\mathcal{R}| = n^2$. Since $|\mathcal{R}|$ decreases every iteration and is non-negative, the loop can be executed at most n^2 times. Hence, the algorithm terminates.

Next, we argue that the above algorithm computes the largest post-fixed point of Φ. Since $\mathcal{R}$ computed by the above algorithm is a fixed point of Φ, we have that $\mathcal{R} \subseteq \Phi(\mathcal{R})$, that is, $\mathcal{R}$ is *post-fixed point* of Φ. Let $\mathcal{S}$ be an arbitrary post-fixed point of Φ, that is, $\mathcal{S} \subseteq \Phi(\mathcal{S})$. We claim that $\mathcal{S} \subseteq \mathcal{R}$ is a loop invariant. We annotate the algorithm as follows.

```
1 R = S × S
2 {S ⊆ R}
3 while Φ(R) ⊂ R
4    {S ⊆ R}
5    R = Φ(R)
6    {S ⊆ R}
7 {S ⊆ R}
```

Initially, $\mathcal{R} = S \times S$ and, hence, the loop invariant $\mathcal{S} \subseteq \mathcal{R}$ holds. In order to conclude that the loop invariant is maintained, it suffices to show that $\mathcal{S} \subseteq \mathcal{R}$ implies $\mathcal{S} \subseteq \Phi(\mathcal{R})$. Assume that $\mathcal{S} \subseteq \mathcal{R}$. Since Φ is monotone, we have that $\Phi(\mathcal{S}) \subseteq \Phi(\mathcal{R})$. Because $\mathcal{S}$ is a post-fixed point of Φ, we have that $\mathcal{S} \subseteq \Phi(\mathcal{S})$.

Hence, $\mathcal{S} \subseteq \Phi(\mathcal{R})$.

5.3 Probabilistic Bisimilarity for Labelled Markov Chains

Kemeny and Snell [KS60] introduced the notion of *lumpability* for Markov chains. This notion was adapted to the setting of labelled Markov chains by Larsen and Skou [LS89] as follows.

Definition 3. *An equivalence relation $\mathcal{R} \subseteq S \times S$ is a probabilistic bisimulation if for all $(s,t) \in \mathcal{R}$,*

- $\ell(s) = \ell(t)$ *and*
- *for all $\mathcal{R}$-equivalence classes C, $\tau(s)(C) = \tau(t)(C)$,*

where $\tau(s)(C) = \sum_{t \in C} \tau(s)(t)$.

As we will show later in this chapter, there exists a largest probabilistic bisimulation, called *probabilistic bisimilarity*. We can compute probabilistic bisimilarity with the partition refinement algorithm discussed in Section 5.2. Let us define the function Φ that is used to refine the partition.

Definition 4. *The function $\Phi : \mathcal{E}(S) \to 2^{S \times S}$ is defined by*

$$\Phi(\mathcal{R}) = \{\, (s,t) \in S \times S \mid \ell(s) = \ell(t) \wedge \tau(s)(C) = \tau(t)(C) \text{ for all } C \in S/\mathcal{R} \,\}.$$

From the above definitions, we can reason that an equivalence relation is a probabilistic bisimulation if and only if it is a post-fixed point of Φ.

Proposition 3. *For all $\mathcal{R} \in \mathcal{E}(S)$, $\mathcal{R}$ is a probabilistic bisimulation if and only if $\mathcal{R} \subseteq \Phi(\mathcal{R})$.*

Proof. Let $\mathcal{R} \in \mathcal{E}(S)$. We prove two implications. First, assume that $\mathcal{R}$ is a probabilistic bisimulation. Let $(s,t) \in \mathcal{R}$. Then $\ell(s) = \ell(t)$ and $\tau(s)(C) = \tau(t)(C)$ for all $C \in S/\mathcal{R}$ according to Definition 3. Thus, $(s,t) \in \Phi(\mathcal{R})$ by Definition 4. Therefore, $\mathcal{R} \subseteq \Phi(\mathcal{R})$.

To prove the opposite implication, assume that $\mathcal{R} \subseteq \Phi(\mathcal{R})$. Let $(s,t) \in \mathcal{R}$. Because $\mathcal{R} \subseteq \Phi(\mathcal{R})$ by assumption, we have that $(s,t) \in \Phi(\mathcal{R})$. Thus, $\ell(s) = \ell(t)$ and $\tau(s)(C) = \tau(t)(C)$ for all $C \in S/\mathcal{R}$ by Definition 4. Therefore $\mathcal{R}$ is a probabilistic bisimulation according to Definition 3.

□

In order to use the conclusions drawn in the previous section, we must prove the following properties of the refinement function Φ, specified in Definition 4. We show that Φ is a function from $\mathcal{E}(S)$ to $\mathcal{E}(S)$ (Proposition 4) and it is monotone (Proposition 5).

Proposition 4. *For all $\mathcal{R} \in \mathcal{E}(S)$, $\Phi(\mathcal{R}) \in \mathcal{E}(S)$.*

Proof. Let $\mathcal{R} \in \mathcal{E}(S)$. We leave to the reader to check that for all s, t, $u \in S$,

- $(s, s) \in \Phi(\mathcal{R})$,
- if $(s, t) \in \Phi(\mathcal{R})$ then $(t, s) \in \Phi(\mathcal{R})$, and
- if $(s, t) \in \Phi(\mathcal{R})$ and $(t, u) \in \Phi(\mathcal{R})$ then $(s, u) \in \Phi(\mathcal{R})$.

Hence, $\Phi(\mathcal{R}) \in \mathcal{E}(S)$. □

Therefore, Φ is a function on the partial order $\mathcal{E}(S)$.

Proposition 5. Φ *is monotone.*

Proof. Let $\mathcal{R}$ and $\mathcal{S}$ be equivalence relations with $\mathcal{R} \subseteq \mathcal{S}$. We have to that prove that $\Phi(\mathcal{R}) \subseteq \Phi(\mathcal{S})$. Let $(s, t) \in \Phi(\mathcal{R})$. Then $\ell(s) = \ell(t)$ and $\tau(s)(C) = \tau(t)(C)$ for all $C \in S/\mathcal{R}$. To conclude that $(s, t) \in \Phi(\mathcal{S})$, it remains to show that $\tau(s)(D) = \tau(t)(D)$ for all $D \in S/\mathcal{S}$. Let $D \in S/\mathcal{S}$. Since $\mathcal{R} \subseteq \mathcal{S}$, the partition into equivalence classes induced by $\mathcal{R}$ is a refinement of the partition induced by $\mathcal{S}$ according to Proposition 1(b). As a consequence, the $\mathcal{S}$-equivalence class D is the disjoint union of $\mathcal{R}$-equivalence classes $C_1, \ldots, C_n$. Hence,

$$\begin{aligned}
\tau(s)(D) &= \tau(s)\left(\bigcup_{1 \leq i \leq n} C_i\right) \\
&= \sum_{1 \leq i \leq n} \tau(s)(C_i) \\
&= \sum_{1 \leq i \leq n} \tau(t)(C_i)
\end{aligned}$$

$$= \tau(t)\left(\bigcup_{1\leq i\leq n} C_i\right)$$

$$= \tau(t)(D).$$

□

Since Φ is monotone (Proposition 5) and probabilistic bisimulations are post-fixed points of Φ (Proposition 3), the partition refinement algorithm presented in the previous section computes the largest probabilistic bisimulation. Thus, we can conclude that there is a largest probabilistic bisimulation termed probabilistic bisimilarity and denoted by $\sim$.

Proposition 6.

(a) $\Phi(S \times S) = \{\, (s,t) \in S \times S \mid \ell(s) = \ell(t) \,\}$.

(b) If $\Phi(\mathcal{R}) \subseteq \mathcal{R} \subseteq \Phi(S \times S)$ *then* $\Phi(\mathcal{R}) = \{\, (s,t) \in S \times S \mid \tau(s)(C) = \tau(t)(C)$ *for all* $C \in S/\mathcal{R} \,\}$.

Proof.

(a) The equivalence relation $S \times S$ has a single equivalence class, namely S. Thus, for all $(s,t) \in S \times S$, we have that $\tau(s)(S) = 1 = \tau(t)(S)$. Hence, in this case, Definition 4 can be simplified to $\Phi(S \times S) = \{\, (s,t) \in S \times S \mid \ell(s) = \ell(t) \,\}$.

(b) Assume that $\Phi(\mathcal{R}) \subseteq \mathcal{R} \subseteq \Phi(S \times S)$. We need to prove that $(s,t) \in \Phi(\mathcal{R})$ if and only if $\tau(s)(C) = \tau(t)(C)$ for all $C \in S/\mathcal{R}$. We prove two implications. Let $(s,t) \in \Phi(\mathcal{R})$. Then by Definition 4, $\tau(s)(C) = \tau(t)(C)$ for all $C \in S/\mathcal{R}$.

Let us now prove the opposite implication. Because $\mathcal{R} \subseteq \Phi(S \times S)$ by assumption, for all $(s,t) \in \mathcal{R}$ we have that $(s,t) \in \Phi(S \times S)$. Therefore, by part (a) we can conclude that $\ell(s) = \ell(t)$ for all $(s,t) \in \mathcal{R}$. Because $\Phi(\mathcal{R}) \subseteq \mathcal{R}$ by assumption, we have that $\ell(s) = \ell(t)$ for all $(s,t) \in \Phi(\mathcal{R})$. Thus, if $\tau(s)(C) = \tau(t)(C)$ for all $C \in S/\mathcal{R}$, then $(s,t) \in \Phi(\mathcal{R})$.

□

The above proposition has the following implications on the partition refinement algorithm to compute probabilistic bisimilarity. According to Proposition 6(a), we can begin with an initial partition in which states with the same labels belong to one equivalence class. Furthermore, during subsequent refinement steps, we need only consider the probabilities of each state transitioning to every equivalence class, as per Proposition 6(b).

Let us apply probabilistic bisimilarity to the labelled Markov chain in Figure 5.1. Firstly, we group states into equivalence classes according to their state labels. State s_0 is the only state labelled with $\{green\}$, hence we place it separately into the first equivalence class. States s_1 and s_2 are both labelled with $\emptyset$, thus we

place them together in the second equivalence class. Similarly, states s_3 and s_4 are both labelled with $\{red\}$ and are thus in the same equivalence class. The resulting partition is shown in Figure 5.3. Secondly, we refine this partition by

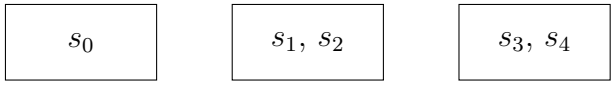


Figure 5.2: The initial partition.

splitting those equivalence classes in which states do not have equal probabilities of transitioning to every equivalence class. The first equivalence class consists of a single state and, therefore, remains unchanged. State s_1 transitions to the second block with probability 0.7 and to the third block with probability 0.3, while state s_2 transitions to the third block with probability 1.0. Thus, we split the second equivalence class into two. States s_3 and s_4 both transition to the third block with probability 1.0, hence we do not split the last equivalence class. The resulting partition is shown in Figure 5.3. No further refinement is possible, thus

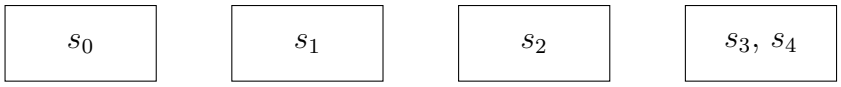


Figure 5.3: The partition following the first refinement.

states s_3 and s_4 are probabilistic bisimilar. The minimized labelled Markov chain, after computing probabilistic bisimilarity, is depicted in Figure 5.4.

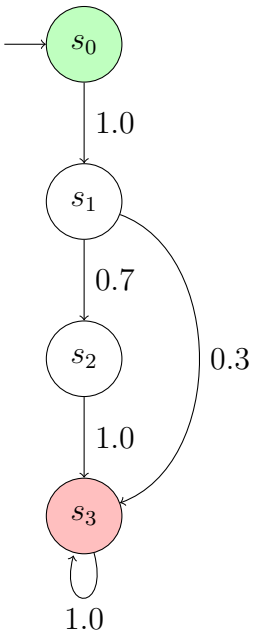


Figure 5.4: The minimized labelled Markov chain with states s_3 and s_4 identified.

6 Computing Probabilistic Bisimilarity

In this chapter, we study four different partition refinement algorithms to compute probabilistic bisimilarity for labelled Markov chains. First we present the $\mathcal{O}(mn)$ algorithm due to Buchholz [Buc00]. Then we present the $\mathcal{O}(m \log n)$ algorithms by Derisavi, Hermanns and Sanders [DHS03] and Valmari and Franceschinis [VF10]. We also briefly discuss the algorithm implemented in PRISM, which is based on the algorithm by Derisavi [Der07]. We use the labelled Markov chain portrayed in Figure 6.1 as the running example.

6.1 Initial Partition

Let us first discuss the initial partition for all of the four partition refinement algorithms mentioned above. According to Proposition 6(a), we can start the partition refinement algorithm with the partition induced by the equivalence relation

$$\{\, (s,t) \in S \times S \mid \ell(s) = \ell(t) \,\}.$$

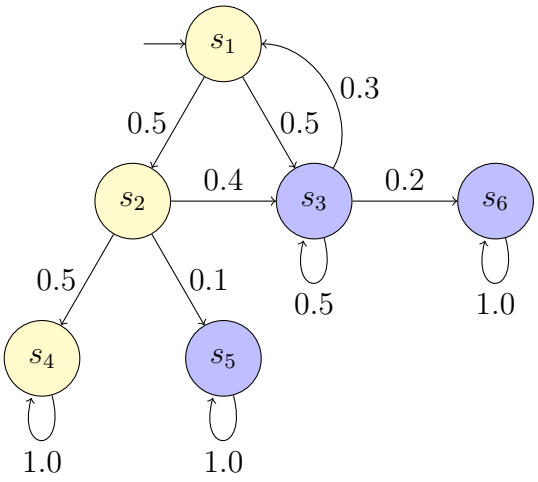


Figure 6.1: A labelled Markov chain.

This partition $\mathcal{I}$ is characterized by

$$\forall B \in \mathcal{I} : \forall s, t \in S : s \in B \wedge t \in B \Leftrightarrow \ell(s) = \ell(t). \tag{6.1}$$

An algorithm to compute this initial partition is described in [BK08, Section 7.3.1]. Here, in Algorithm 6, we give an alternative presentation that is used in PRISM.[8]

For $a \in L$, in line 1–8 we compute the set B_a defined by

$$B_a = \{\, s \in S \mid a \in \ell(s) \,\}.$$

The set $\mathcal{B}$ is implemented as a list and, hence, each $B \in \mathcal{B}$ has an index which is used in line 24. The variable n keeps track of the number of blocks in $\mathcal{B}$. In the loop of line 11–21, for each $a \in L$, each block $B \in \mathcal{B}$ is refined as $B \cap B_a$ and $B \setminus B_a$. The

[8]See the class `explicit.Bisimulation` of the PRISM distribution which can be found at the URL github.com/prismmodelchecker/prism.

Algorithm 6: initialPartition(ℓ)

```
   Input: ℓ ∈ S → 2^L the labelling function
 1 foreach a ∈ L do
 2     B_a ← ∅
 3 end
 4 foreach s ∈ S do
 5     foreach a ∈ ℓ(s) do
 6         B_a ← B_a ∪ {s}
 7     end
 8 end
 9 B ← {S}
10 n ← 1
11 foreach a ∈ L do
12     foreach B ∈ B do
13         if B_a ∩ B ≠ ∅ then
14             B ← B \ {B} ∪ {B ∩ B_a}
15             if B \ B_a ≠ ∅ then
16                 B ← B ∪ {B \ B_a}
17                 n ← n + 1
18             end
19         end
20     end
21 end
22 foreach B ∈ B do
23     foreach s ∈ B do
24         partition[s] ←  index of B
25     end
26 end
```

conditions in line 13 and 15 ensure that the empty set is not added to $\mathcal{B}$. Assuming that label a_k is considered in the kth iteration of the loop of line 11–21, then the loop maintains the following invariant:

$$\forall B \in \mathcal{B} : \forall s, t \in S : s \in B \wedge t \in B \Leftrightarrow \ell(s) \cap \{a_0, \ldots, a_k\} = \ell(t) \cap \{a_0, \ldots, a_k\}.$$

Hence, (6.1) holds once we reach line 22.

The running time of the above algorithm is $\Theta(|S| \cdot |L|)$ (see, for example, [BK08, Lemma 7.33]). In the PRISM implementation, the partition $\mathcal{B}$ is represented as `ArrayList<BitSet>`.

If we apply Algorithm 6 to the labelled Markov chain illustrated in Figure 6.1, we obtain an initial partition with two blocks. States with the same set of labels are grouped in a single block. States s_1, s_2 and s_4 are labelled with $\{yellow\}$ and are thus placed in the first block, while the rest of the states s_3, s_5 and s_6 are labelled with $\{blue\}$ and are placed in the second block. The initial partition $\mathcal{B}$ is shown in Figure 6.2.

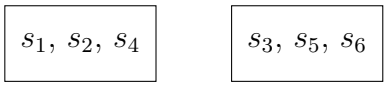


Figure 6.2: The initial partition.

6.2 Buchholz

Buchholz's algorithm computes probabilistic bisimilarity for stochastic automata [Buc00]. We adapt his algorithm for labelled Markov chains. Let us first recall the definition of a stochastic automaton.

Definition 5. *A stochastic automaton is a tuple* $\langle S, A, R, E, \rho \rangle$ *consisting of*

- *a nonempty and finite set S of states,*
- *a nonempty and finite set A of actions,*
- *a nonempty and finite set R of rewards,*
- *for each $a \in A \cup \{i\}$, a transition rates matrix $E_a : (S \times S) \to \mathbb{R}$ such that for all s, $t \in S$, $E_a(s,t) \geq 0$, and*
- *for each $r \in R$, a reward vector $\rho_r : S \to \mathbb{R}$ such that for all $s \in S$, $\rho_r(s) > 0$.*

The transitions of a stochastic automaton are labelled with actions. The transitions labelled with the action i are internal transitions. For a state s and an action a, if $\sum_{t \in S} E_a(s,t) = 0$ then state s does not have any transitions labelled with a. Otherwise, that is, if $\sum_{t \in S} E_a(s,t) > 0$, we can translate the transition rates into transition probabilities in the following standard way to extract the embedded Markov chain (see, for example, [Gal13, p. 327]):

$$P_a(s,t) = \frac{E_a(s,t)}{\sum_{t \in S} E_a(s,t)}.$$

A labelled Markov chain $\langle S, L, \tau, \ell \rangle$ can be viewed as a stochastic automaton as follows. The labelled Markov chain and the stochastic automaton have the same set of states S. Since the transitions of a labelled Markov chain are not labelled, we only consider a single action a with which all transitions of the stochastic automaton are labelled. Thus, the set of actions $A = \{a\}$. The transition rates of the transitions

labelled with the action a correspond to τ, that is, for all s, $t \in S$, $E_a(s,t) = \tau(s)(t)$. Note that, since for all $s \in S$, $\sum_{t \in S} \tau(s)(t) = 1$, we can conclude that $P_a(s,t) = \tau(s)(t)$ for all s, $t \in S$. We do not consider internal transitions in the stochastic automaton. Thus for all s, $t \in S$, $E_i(s,t) = 0$, that is, the matrix of internal transition rates is filled with zeros and can be disregarded. Similarly, we only consider a single reward r, consequently we have a single reward vector ρ_r. Thus, the labelling function ℓ can be captured by the reward vector ρ_r as follows. Assume that the set $\{\, \ell(s) \mid s \in S \,\}$ has M elements. Let $\iota : \{\, \ell(s) \mid s \in S \,\} \to \{\, m \in \mathbb{N} \mid 1 \leq m \leq M \,\}$ be a bijection. Then, for $s \in S$, define $\rho_r(s) = \iota(\ell(s))$. Note that for all s, $t \in S$, $\rho_r(s) = \rho_r(t)$ if and only if $\ell(s) = \ell(t)$.

We now show that if you take an equivalence relation on the states, such a relation is a probabilistic bisimulation for the labelled Markov chain if and only if it is a probabilistic bisimulation for the induced stochastic automaton. Let us first define probabilistic bisimulation for stochastic automata.

Definition 6. *An equivalence relation $\mathcal{R} \subseteq S \times S$ is a probabilistic bisimulation if for all $(s,t) \in \mathcal{R}$,*

- *for each $r \in R$, $\rho_r(s) = \rho_r(t)$ and*
- *for each $a \in A \cup \{i\}$, for all $\mathcal{R}$-equivalence classes C, $E_a(s,C) = E_a(t,C)$,*

where $E_a(s,C) = \sum_{t \in C} E_a(s,t)$.

As mentioned above, we have that $R = \{r\}$ and for all s, $t \in S$, $\rho_r(s) = \rho_r(t)$ if and only if $\ell(s) = \ell(t)$. Furthermore, since we have that $A = \{a\}$, the transition rates of the transitions labelled with this action a correspond to τ, that is, for all s, $t \in S$, $E_a(s,t) = \tau(s)(t)$. Moreover, for all s, $t \in S$, $E_i(s,t) = 0$. Thus we can conclude that Definition 6 coincides with the notion of probabilistic bisimulation introduced earlier for labelled Markov chains in Definition 3.

6.2.1 The Algorithm

We demonstrated how a labelled Markov chain can be seen as a special version of a stochastic automaton. Thus, we observe that rather than taking a labelled Markov chain and turning it into a stochastic automaton, we specialize the algorithm of Buchholz to this specific type of stochastic automaton. In the algorithm, both states and equivalence classes are identified by positive integers. Let us define the variables used in the algorithm. A splitter, X, is an equivalence class by which all equivalence classes are refined according to their transition probabilities into the splitter. We use an array *val* of reals, where $val[s]$ is the probability of the state (with index) s transitioning to the splitter X. I is the set of the indices of those equivalence classes which are potential splitters. For state s, the array cell $states[s]$ contains the index of the equivalence class to which s belongs. For equivalence class c, the array cell $class[c]$ contains the set of state indices belonging

to c. Note that the integer N is always equal to the size of *class*, but we introduce this additional variable for convenience. Finally, for equivalence class c, the array cell $split[c]$ contains three values. The boolean value $split[c].init$ captures whether a state belonging to c has already been processed in the current iteration. The real value $split[c].val$ is the probability of each state in c transitioning to the current splitter. Lastly, if the integer value $split[c].next$ is a non-zero value, it denotes the index of the first newly created equivalence class when c is split in the current iteration, otherwise a value of zero indicates that equivalence class c has not been split yet. Since each equivalence class has a reference to the next equivalence class, if it exists, this creates a linked list of equivalence classes that form a partition of c.

The specialized algorithm is presented in Algorithm 7. In line 2 we initialize the set I to include all equivalence classes in the initial partition. In lines 4–8 we initialize each component of *split* by setting *init* to *false* and the values *val* and *next* to 0. In lines 9–16 we choose a splitter from the set of potential splitters and then calculate the probability of each state transitioning to that splitter. The method split in line 17 refines the current partition and is defined in Algorithm 8.

The refinement step, that is, the method split shown in Algorithm 8, is different from that introduced in Proposition 6(b), since we only consider a single equivalence class as a splitter during each iteration instead of all equivalence classes. Neverthe-

Algorithm 7: Buchholz(τ, *class*, *states*)

Input: $\tau \in S \to \mathcal{D}(S)$ the transition function
class $\in \{1, \ldots, N\} \to \mathcal{P}(S)$ the set of states in each block of the initial partition
states $\in S \to \{1, \ldots, N\}$ the block each state belongs to in the initial partition

```
1  N ← size of class
2  I ← {1, ..., N}
3  while I ≠ ∅ do
4      for c ← 1 to N do
5          split[c].init ← false
6          split[c].val ← 0
7          split[c].next ← 0
8      end
9      choose an element X from I
10     I ← I \ {X}
11     foreach s ∈ S do
12         val[s] = 0
13         foreach t ∈ X do
14             val[s] ← val[s] + τ(s)(t)
15         end
16     end
17     N ← split(val, I, N, states, split, class)
18 end
```

less, the algorithm computes probabilistic bisimilarity as proved in [Buc00], because we keep track of the equivalence classes which must still be used as splitters, in the set I. Algorithm 8 is explained in detail in the paper by Buchholz.

We implemented the algorithm in Java, using the following data structures. To store the transition function τ, we use a well known sparse matrix representation known as a list of lists. For each state s, the list contains a list that represents $\tau(s)$. The elements of the list representing $\tau(s)$ are those pairs $(t, \tau(s)(t))$ for

Algorithm 8: split(val, I, N, $states$, $split$, $class$)

Input: $val \in S \to [0,1]$ the probability of each state transitioning to the splitter X
$I \subseteq \{1,\dots,N\}$ the set of indices of potential splitters
$N \in \mathbb{N}$ the number of blocks
$states \in S \to \{1,\dots,N\}$ the block each state belongs to
$split \in \{1,\dots,N\} \to \{true, false\} \times [0,1] \times \mathbb{N}_0$ stores information about each block
$class \in \{1,\dots,N\} \to \mathcal{P}(S)$ the set of states in each block

1 **foreach** $s \in S$ **do**
2 $c \leftarrow states[s]$
3 **if** $\neg\ split[c].init$ **then**
  `//` s `belongs to` c `and` c `has not been initialized yet`
4 $class[c] \leftarrow \{s\}$
5 $split[c].init \leftarrow true$
6 $split[c].val \leftarrow val[s]$
7 **else**
8 **if** $split[c].val \neq val[s] \wedge split[c].next = 0$ **then**
  `//` s `does not belong to` c `and` c `has not been split`
9 $I \leftarrow I \cup \{c\}$
10 **end**
11 **while** $split[c].val \neq val[s] \wedge split[c].next \neq 0$ **do** $c \leftarrow split[c].next$
12 **if** $split[c].val = val[s]$ **then**
  `//` s `belongs to` c
13 $states[s] \leftarrow c$
14 $class[c] \leftarrow class[c] \cup \{s\}$
15 **else**
  `//` s `belongs to a new block`
16 $N \leftarrow N + 1$
17 $I \leftarrow I \cup \{N\}$
18 $split[c].next \leftarrow N$
19 $states[s] \leftarrow N$
20 $class[N] \leftarrow \{s\}$
21 $split[N].init \leftarrow true$
22 $split[N].val \leftarrow val[s]$
23 **end**
24 **end**
25 **end**
26 **return** N

which $\tau(s)(t) \neq 0$. For the dynamic arrays *class* and *split*, we use an `ArrayList`. For the values in the array *class*, we use a `HashSet`, while for the array *split*, we create a private inner-class with `boolean` attribute *init*, `double` attribute *val* and `int` attribute *next*.

6.2.2 An Example

Let us apply the algorithm to the labelled Markov chain illustrated in Figure 6.1. The initial partition is shown in Figure 6.2. Assume that the second equivalence class is picked as the splitter X. Using the formula, $val[s] = \sum_{t \in X} \tau(s)(t)$ which corresponds to lines 11–16 in Algorithm 7, the array *val* is calculated as [0.5, 0.5, 0.7, 0.0, 1.0, 1.0].

In order to visualize the execution of the method split, we depict equivalence classes as follows. Equivalence classes that have not yet been initialized, that is, $init = false$, are coloured grey, while initialized equivalence classes are coloured white. The value on the top right of an equivalence class is the probability of the states in that class transitioning to the splitter X, that is, the value of *val*. If the value of *next* is a non-zero value, we draw a pointer to the next equivalence class. Equivalence classes with a thick border are not included in the set I and may not be chosen as a splitter again. Figure 6.3 depicts the changes in the blocks of the partition during the first refinement at each iteration of the splitting method.

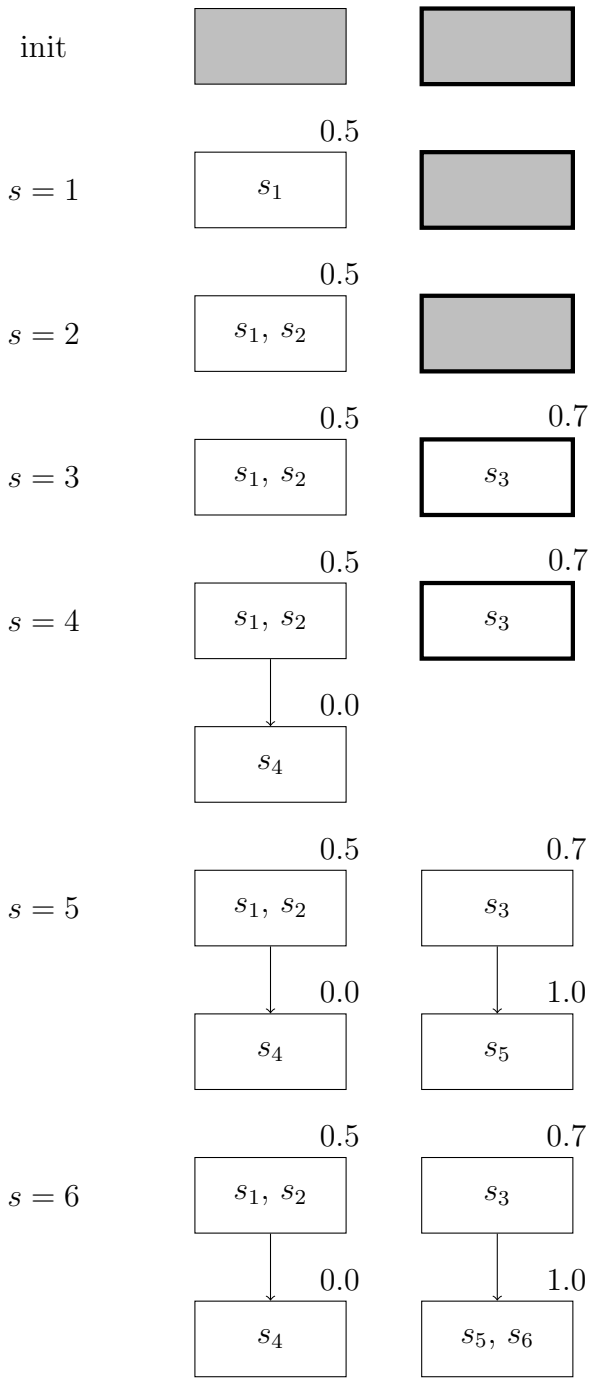


Figure 6.3: The first execution of the method split.

At first, none of the blocks are initialized. As per line 1 of Algorithm 8, we iterate through the states sequentially, attempting to place them in the equivalence class they belonged to in the previous partition, that is, the initial partition shown in Figure 6.2, and splitting if necessary. States s_1 and s_2 are both placed in the first equivalence class as $val[1] = val[2] = 0.5$. State s_3 is then placed in the second equivalence class. We attempt to place state s_4 in the first equivalence class, but $val[4] = 0.0 \neq 0.5$, so we split the equivalence class, creating a third equivalence class which we initialize and then put state s_4 in it. Similarly, we attempt to place state s_5 in the second equivalence class, however $val[5] = 1.0 \neq 0.7$, thus we create a fourth equivalence class. Since the second equivalence class was split, we re-add it to the set of future splitters I. Lastly, we attempt to place state s_6 in the second equivalence class, but since $val[6] = 1.0 \neq 0.7$, we attempt to place state s_6 in the next equivalence class, namely the fourth equivalence class, and are successful. The next partition is shown in Figure 6.4.

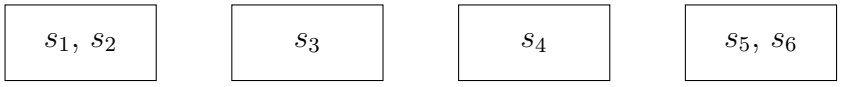


Figure 6.4: The resulting partition.

Assume that during the second refinement, the fourth equivalence class is picked as the splitter X. Then we compute $val = [0.0, 0.1, 0.2, 0.0, 1.0, 1.0]$. Figure 6.5 illustrates the execution of the method split in line 17 of Algorithm 7. State s_1 is

placed into the first equivalence class. Since $val[2] = 0.1 \neq 0.0$, we create a fifth equivalence class for state s_2. The remaining states s_3, s_4, s_5 and s_6 are then placed in the same equivalence classes as in Figure 6.4 without any conflicts.

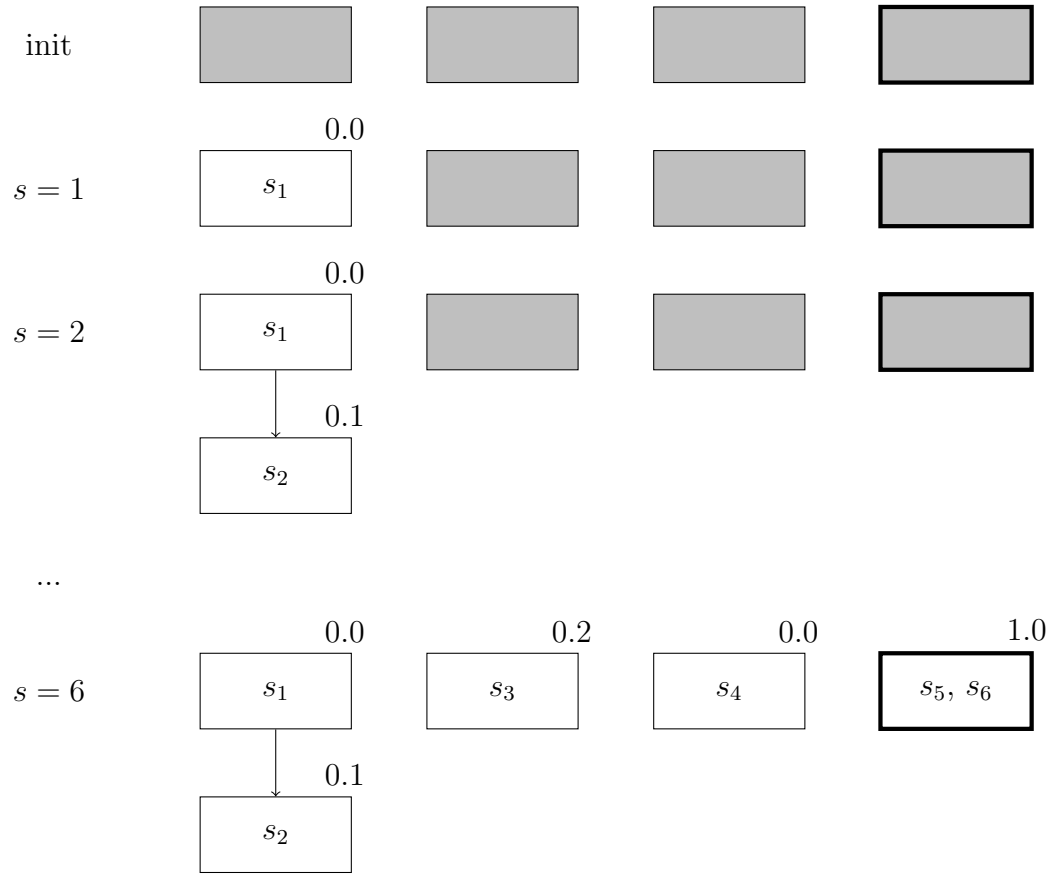


Figure 6.5: The second execution of the method split.

Four more subsequent refinements occur, with the first, second, third and fifth equivalence classes as splitters; however, none of the equivalence classes are split further. Hence, the final partition is presented in Figure 6.6. Thus, states s_5 and s_6 are probabilistic bisimilar.

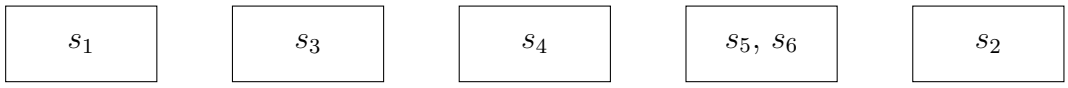


Figure 6.6: The final partition.

6.2.3 Errors

In Algorithm 8, the `else` block on lines 15–23 is only executed when state s does not belong to equivalence class c, that is $split[c].val \neq val[s]$, and $split[c].next$ does not point to another equivalence class, thus a new equivalence class N must be created. $split[c].next$ is then set to N in line 18 and the state s is added to equivalence class N in lines 19 and 20. In line 21 we initialize this new equivalence class N and in line 22 we set the value $split[N].val$ to $val[s]$, as it represents the probability of each state in N, namely state s, transitioning to the splitter X. Conversely, in the paper, lines 21 and 22 of Algorithm 8 are erroneously written as follows instead.

21 $split[c].init \leftarrow true$
22 $split[c].val \leftarrow val[s]$

However, this means that the new equivalence class N is not initialized even though a state has been added to the equivalence class. Moreover, the value $split[N].val$ is not set and thus has the default value 0, which would incorrectly denote that state s does not have an outgoing transition to X if $val[s] \neq 0$. Furthermore, the value of $split[c].val$ is modified, thus, denoting an incorrect value for the probability with which each state in equivalence class c transitions to X.

Below we provide a counterexample in which states are falsely reported as distinct when using the original algorithm. Consider the labelled Markov chain in Figure 6.7.

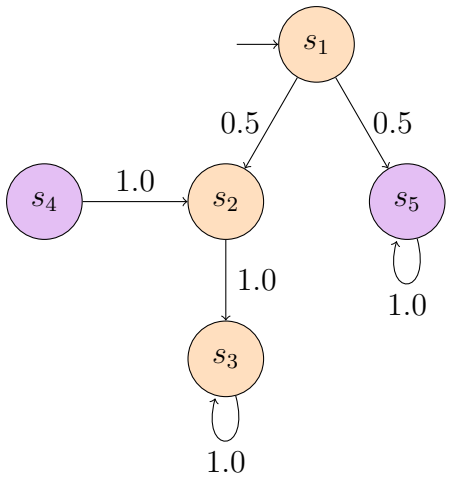


Figure 6.7: A labelled Markov chain.

We generate the initial partition by placing all states labelled with $\{orange\}$ in the first equivalence class and all states labelled with $\{purple\}$ in the second equivalence class. The initial partition is shown in Figure 6.8.

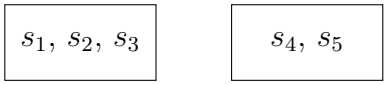


Figure 6.8: The initial partition.

Assume that the first equivalence class is picked as the splitter X, then $val = [0.5, 1.0, 1.0, 1.0, 0.0]$. Figure 6.9 depicts the changes to the partition during the first refinement at each iteration of the original splitting method. We show variables

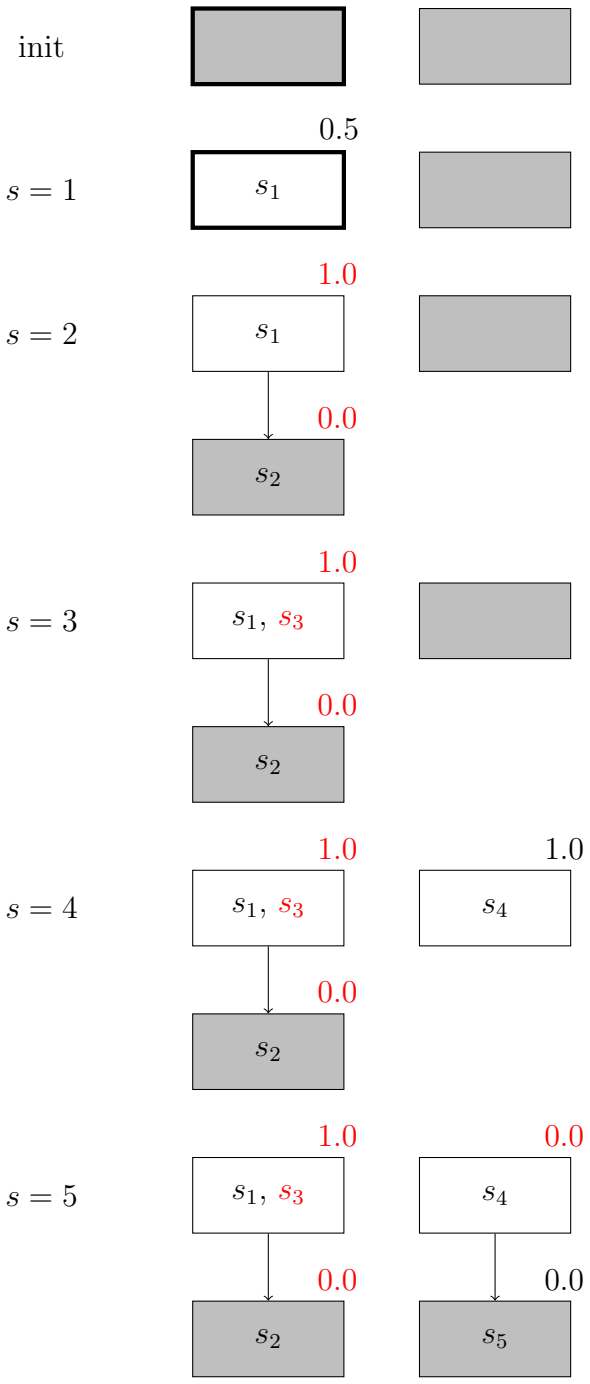


Figure 6.9: The first execution of the original method split.

assigned with incorrect values and states placed in incorrect equivalence classes in red.

State s_1 is placed into the first equivalence class. Since $val[2] = 1.0 \neq 0.5$, we split the first equivalence class and place state s_2 in the new equivalence class. The probability of each state of the first equivalence class transitioning to the splitter, that is, $split[1].val$, is incorrectly modified from 0.5 to 1.0, while the new equivalence class is left with the default value of 0.0 instead of being assigned to 1.0. Thus, when we attempt to place state s_3 in the first equivalence class, we are successful as $val[3] = 1.0$. However, states s_1 and s_3 do not have the same probability of transitioning to the splitter, that is $val[1] \neq val[3]$, thus these states should not belong to the same equivalence class. State s_3 should be placed in the third equivalence class with state s_2. State s_4 is placed in the second equivalence class. We split the second equivalence class and place state s_5 in the new equivalence class, as $val[5] = 0.0 \neq 1.0$. Similarly, the value $split[2].val$ is incorrectly modified from 1.0 to 0.0. Notice also that none of the newly created equivalence classes are initialized. The resulting partition is shown in Figure 6.10.

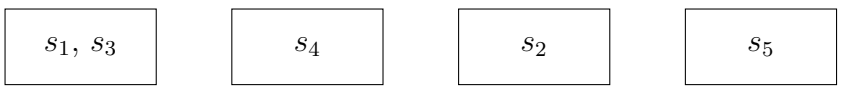


Figure 6.10: The resulting partition.

Assume that the first equivalence class is picked as the splitter again, then

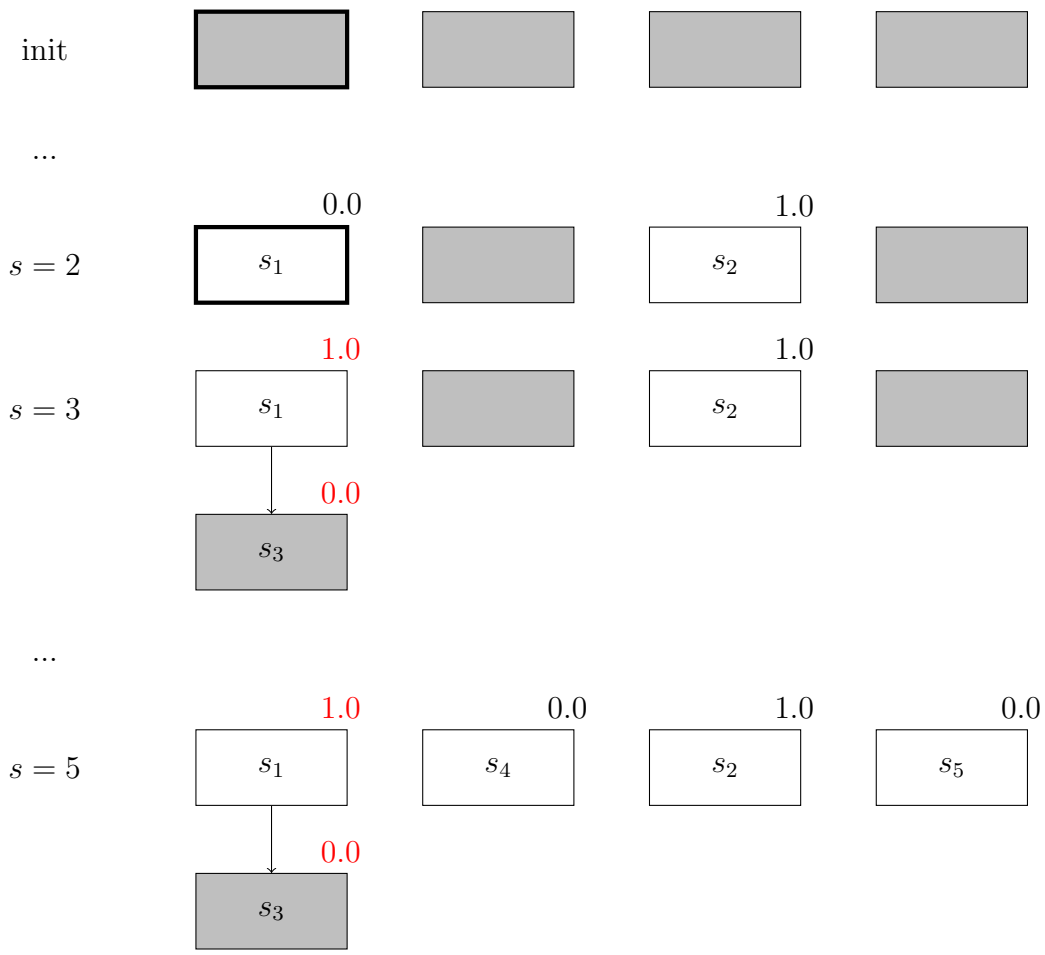


Figure 6.11: The second execution of the original method split.

$val = [0.0, 1.0, 1.0, 0.0, 0.0]$. Figure 6.11 depicts a summary of the changes to the partition during the second refinement. States s_1 and s_2 are placed into their original equivalence classes, that is, the first and third equivalence classes, respectively. We attempt to place state s_3 into the first equivalence class as well, but since $val[3] = 1.0 \neq 0.0$, we split the first equivalence class and place state s_3 in the newly created equivalence class. Once again, the value $split[1].val$ is incorrectly modified

to 1.0 instead of $split[5].val$. We then place state s_4 in the second equivalence class and state s_5 in the fourth equivalence class without any conflict.

The resulting partition is shown in Figure 6.12. Since all of the equivalence classes are singletons, the partition can not be refined further and this is the final partition. However, this is the wrong result as states s_2 and s_3 belong to different equivalence classes but are in fact probabilistic bisimilar.

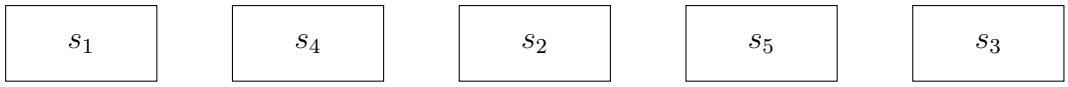


Figure 6.12: The incorrect final partition.

We corrected lines 21–22 of the algorithm as shown in Algorithm 8. In Figure 6.13, we show the execution of the modified method on the initial partition depicted in Figure 6.8. Evidently, the errors discussed during the execution of the algorithm in Figure 6.9 are no longer present and the initial partition is correctly refined. The four subsequent refinement steps do not alter the resulting partition. The final partition is displayed in Figure 6.14. States s_2 and s_3 are identified as probabilistic bisimilar.

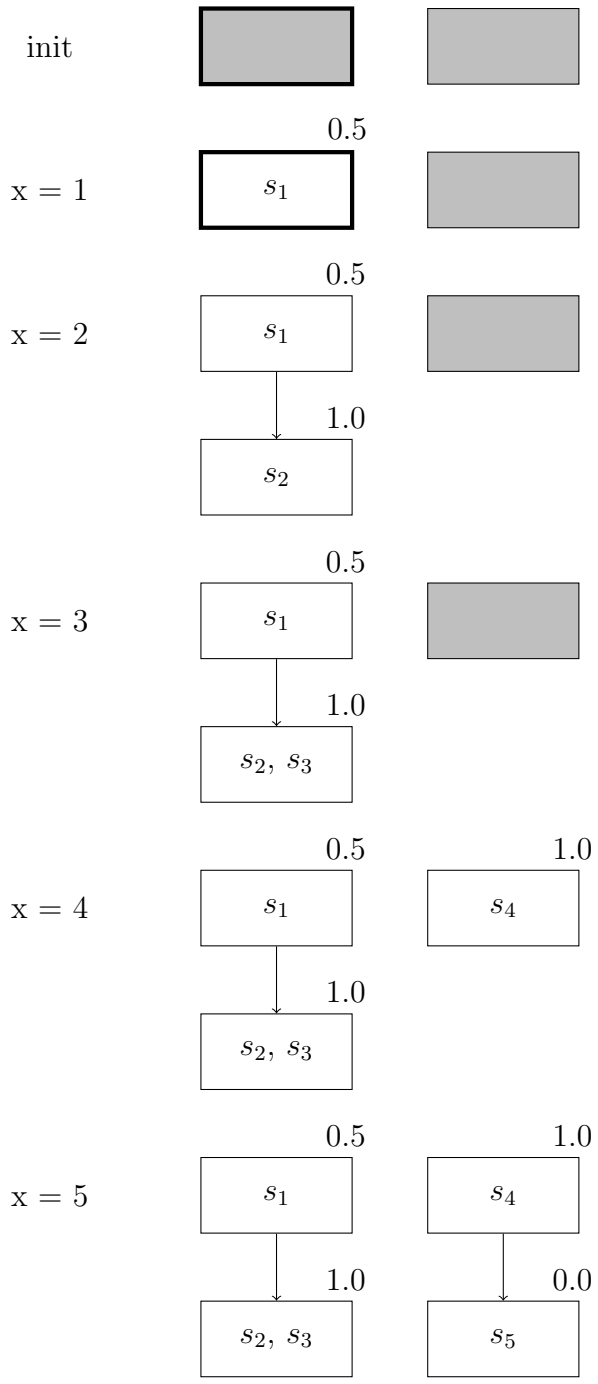


Figure 6.13: The execution of the modified method split.

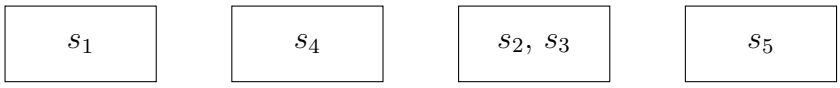


Figure 6.14: The correct final partition.

6.3 Derisavi, Hermanns and Sanders

6.3.1 The Algorithm

The algorithm developed by Derisavi, Hermanns and Sanders computes probabilistic bisimilarity for Markov chains [DHS03]. The algorithm, presented in Algorithm 9, takes two inputs, namely the transition function τ of the Markov chain and an initial partition of the state space P. We use the algorithm to compute probabilistic bisimilarity for labelled Markov chains by providing an initial partition in which all states with the same labels are in the same block, as per Proposition 6(a). Note that a version of this algorithm [KKZJ07] has been implemented in the model checker MRMC [KKZ05].

Algorithm 9: DHS(τ, P)

Input: $\tau \in S \to \mathcal{D}(S)$ the transition function
$P \in \mathcal{P}(S)$ the initial partition of the state space S

1 $I \leftarrow$ the set of all blocks in P
2 **while** $I \neq \emptyset$ **do**
3 choose an element X from I
4 $I \leftarrow I \setminus \{X\}$
5 split(X, P, I, τ)
6 **end**

Algorithm 10: split(X, P, I, τ)

Input: $X \in P$ the splitter
$P \in \mathcal{P}(S)$ the current partition
$I \subseteq P$ the set of potential splitters
$\tau \in S \to \mathcal{D}(S)$ the transition function

```
1  L ← ∅
2  B ← ∅
3  foreach t ∈ X do
4      foreach s ∈ t.predecessors do  s.sum ← 0
5  end
6  foreach t ∈ X do
7      foreach s ∈ t.predecessors do
8          s.sum ← s.sum + τ(s)(t)
9          L ← L ∪ {s}
10     end
11 end
12 foreach s ∈ L do
13     c ← s.block
14     delete s from c
15     c_T.insert(s)
16     B ← B ∪ {c}
17 end
18 foreach c ∈ B do
19     subblocks ← the set of all blocks in c_T
20     foreach d ∈ subblocks do add d to P
21     if c ∈ I then
22         I ← I ∪ subblocks
23     else
24         c_L ← max({c} ∪ subblocks)
25         I ← (I ∪ {c} ∪ subblocks) \ {c_L}
26     end
27     c_T.clear()
28 end
```

I is the set of blocks which are potential splitters, while X is the current splitter. The method split in line 5 is defined in Algorithm 10. The set L contains

states which must be processed, that is, those states which are predecessors to X. Similarly, the set B contains the blocks which must be processed, that is, those blocks to which the states in L belong. For state s, the real value $s.sum$ is the probability of the state transitioning to X, the block $s.block$ is the block to which the state belongs, and the set $s.predecessors$ is the set of states which have a non-zero probability of transitioning to state s. In the algorithm, splay trees [ST85] are used to refine blocks of the partition. Each block c, has a corresponding splay tree, denoted as c_T. Each node in c_T stores a subblock of c and the probability of each state in this subblock transitioning to the current splitter X. When a state s is inserted into the splay tree c_T, it is placed into the block of the node associated with the probability $s.sum$ if it exists, otherwise a new node is created. $s.block$ is updated to reference the block in which it is placed. Finally, the block c_L denotes the largest block among c and all its subblocks in the splay tree c_T.

We implemented the algorithm in Java, using the following data structures. The current partition P and the set of potential splitters I are implemented by `LinkedList` with elements of type `Block`. For the two sets L and B, we use the class `HashSet`. States are represented by a private inner-class with an `int` attribute *ID*, `Block` attribute *block* and `LinkedHashMap` attribute *predecessors*. For a state s, the attribute $s.predecessors$ acts as an adjacency list, by storing predecessor states as keys and the probability of transitioning from a specific predecessor state to s as the

values. In this way, we do not need to store the transition function τ. Lastly, we also represent blocks by a private inner-class with an `int` attribute *ID*, `HashSet` attribute *elements* and `SplayTree` attribute *tree*. For a block c the attribute $c.elements$ is the set of states which belong to c and the attribute $c.tree$ stores the subblocks of c during a refinement step, denoted as c_T in the algorithm. We implemented the class `SplayTree` according to the algorithm by Sleater and Tarjan [ST85].

6.3.2 An Example

Let us apply the algorithm to our running example, that is, the labelled Markov chain illustrated in Figure 6.1. The initial partition is depicted in Figure 6.2. Assume that the second block is picked as the splitter X. We calculate $s.sum = \sum_{t \in X} \tau(s)(t)$, for each s that is a predecessor of some $t \in X$, which corresponds to lines 3–11 in Algorithm 10.

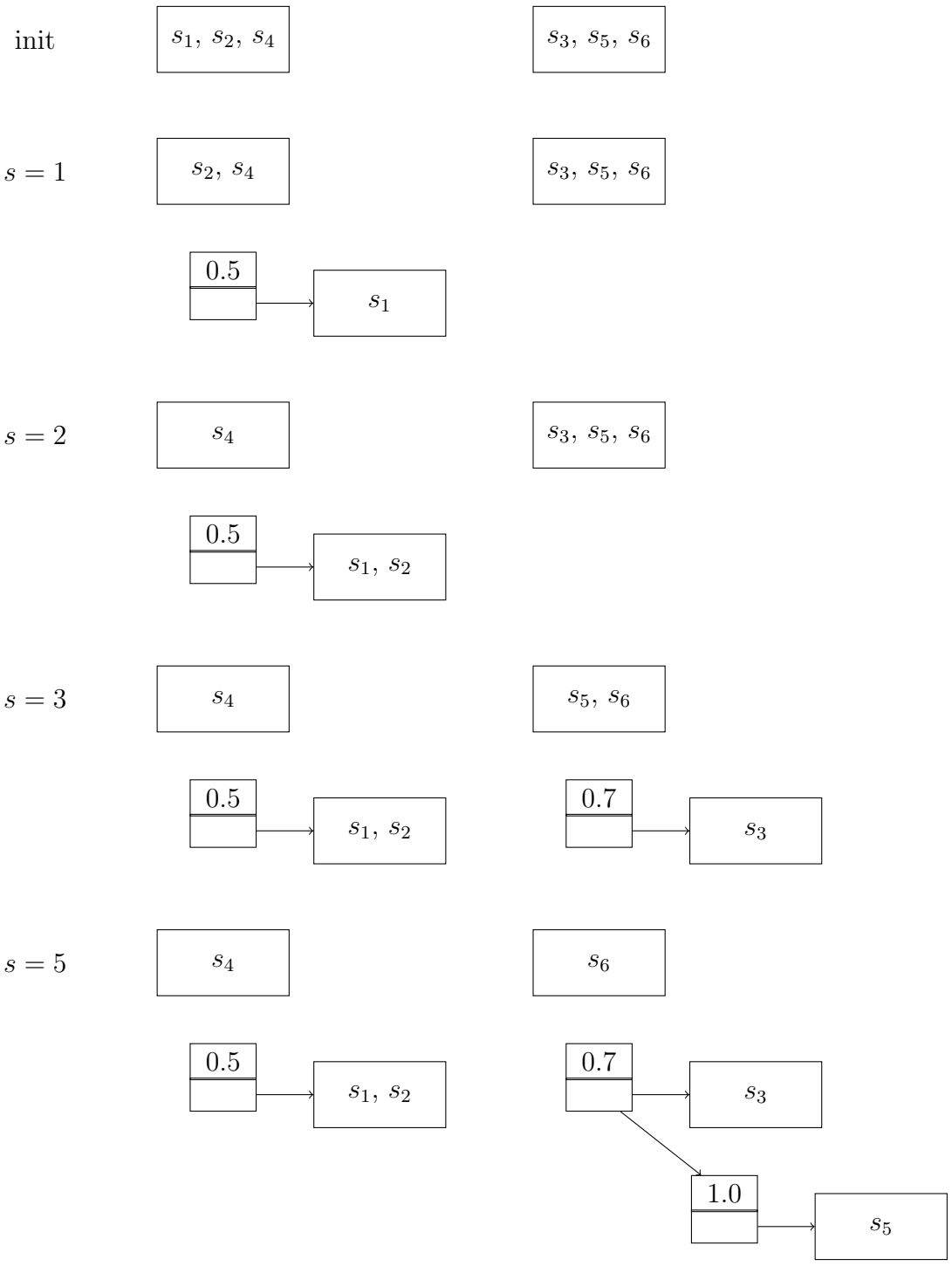
1 $s_1.sum = 0.5$
2 $s_2.sum = 0.5$
3 $s_3.sum = 0.7$
4 $s_5.sum = 1.0$
5 $s_6.sum = 1.0$

Observe that state s_4 is not a predecessor of the second block X, thus the probability of s_4 transitioning to the splitter is 0 and we do not need to calculate it.

In order to visualize the execution of the method split, if the splay tree associated with a block c is not empty, we display the splay tree c_T directly below it. A node

in the splay tree consists of a reference to a block and the probability with which the states in the block transition to the splitter X. Figure 6.15 depicts the changes in the blocks of the partition during the first refinement at each iteration of the method split. If the splitter is not split, it cannot be chosen as the splitter again. Furthermore, the largest sub-block of an original block that is not in the set I may not be used as a splitter. These equivalence classes are drawn with a thick border are not included in the set I.

We begin with the initial partition and iterate through the predecessors of the splitter sequentially, as per line 12 of Algorithm 10. We remove each predecessor state and insert it into the splay tree associated with the block it belongs to. Let us consider the first equivalence class. State s_1 is placed in the block at the root of the splay tree, since the tree was empty. State s_2 is also added to the block at the root of the splay tree, as $s_1.sum = s_2.sum = 0.5$. Observe that since state s_4 is not a predecessor of the splitter, it remains in its original block and we do not insert it into the splay tree. Let us now consider the second equivalence class. State s_3 is placed in the block at the root of the tree. When we insert state s_5 into the splay tree, we cannot place it in the root since $s_5.sum = 1.0 \neq 0.7$, thus, since $1.0 > 0.7$ we create a new node that is the right child of the root. Lastly, state s_6 in placed in the same block as state s_5 as they have the same probability of transitioning to the splitter.


init
$s_1,\ s_2,\ s_4$
$s_3,\ s_5,\ s_6$
$s = 1$
$s_2,\ s_4$
$s_3,\ s_5,\ s_6$
0.5
s_1
$s = 2$
s_4
$s_3,\ s_5,\ s_6$
0.5
$s_1,\ s_2$
$s = 3$
s_4
$s_5,\ s_6$
0.5
$s_1,\ s_2$
0.7
s_3
$s = 5$
s_4
s_6
0.5
$s_1,\ s_2$
0.7
s_3
1.0
s_5

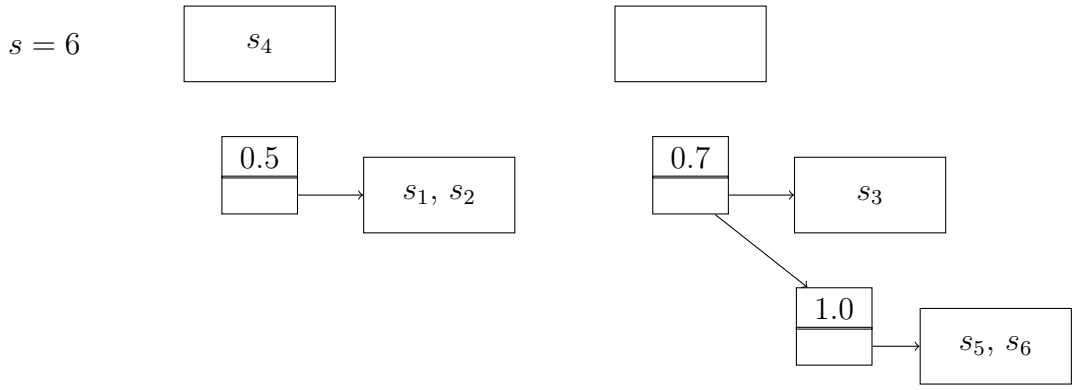


Figure 6.15: The first execution of the method split.

We now iterate through the blocks which contained a predecessor to the splitter X, as per line 18. The first block was not yet used as a splitter, thus, it is part of the set I and we add all of its subblocks to I as well in line 22. The second block was used as a splitter and is no longer part of the set I, hence we add the second block and all of its subblocks to I, except the largest block among them, that is, $\{s_5,\ s_6\}$, according to lines 24–25. The next partition is shown in Figure 6.16.

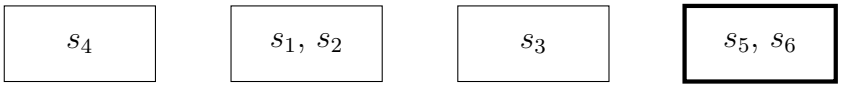


Figure 6.16: The resulting partition.

Assume that during the second refinement, the first block is picked as the splitter X. The state s_4 has two predecessors for which we compute the following.

1 $s_2.sum = 0.5$
2 $s_4.sum = 1.0$

Figure 6.17 illustrates a simplified version of the execution of the method split. States s_2 and s_4 are placed in the blocks of the roots of their respective trees, since the splay trees were empty. The blocks $\{s_3\}$ and $\{s_5, s_6\}$ are not displayed, since they do not contain any states that are predecessors of state s_4.

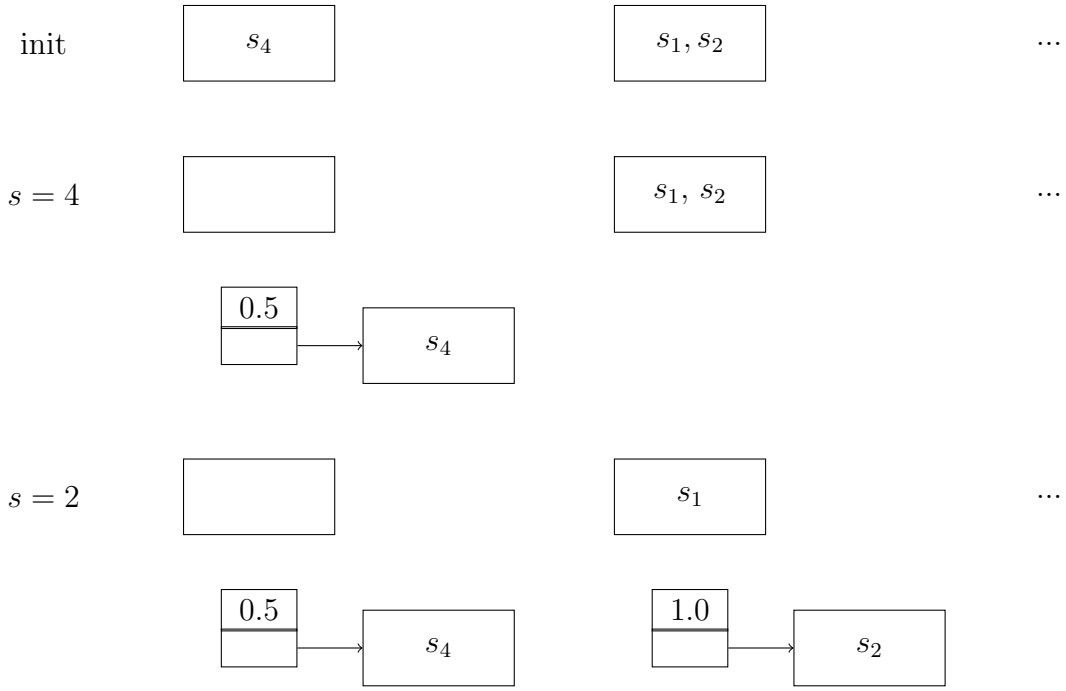


Figure 6.17: The second execution of the method split.

The splitter X is not split and, therefore, cannot be used as a splitter again. Three more subsequent refinements occur, with the second, third and fourth equivalence classes as splitters. None of the equivalence classes are split further; hence, the final partition is presented in Figure 6.18, which corresponds to the final partition obtained by Buchholz's algorithm in Section 6.2.2.

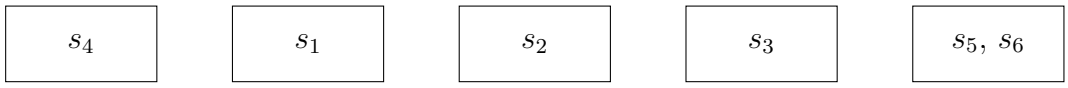


Figure 6.18: The final partition.

6.3.3 Errors

As specified in lines 21–26 of Algorithm 10, we exclude the largest subblock c_L from the set of future splitters, if the original block c is not a potential splitter. This strategy was introduced by Hopcroft [Hop71]. Since the sum of the probabilities of the outgoing transitions of a state equal to one, excluding a single block of the partition from the set of splitters does not result in a loss of information. The probability of any state transitioning to the excluded block is equal to one minus the sum of the probabilities of that state transitioning to the rest of the blocks, thus, if two states s and t have equal probabilities of transitioning to each other block in the partition, they will also have equal probability of transitioning to the excluded block.

If a block c has been used as a splitter, and two states s and t belong the same block, we know that they have equal probability of transitioning to c. Thus, we can exclude the largest subblock c_L of c from the set of future splitters, since if s and t are not split by any of the other subblocks of c, then they will not be split by c_L as described above. However, this is not true if we exclude the largest subblock c_L

when c has not been used as a splitter yet. Two states s and t may be in the same block but have different probabilities of transitioning to c, thus, they may not be split by any of the other subblocks of c except c_L. Hence, if a block c is a potential splitter, we must add all subblocks of c to the set of future splitters.

In the paper [DHS03], the largest subblock c_L is always excluded from the set of future splitters, regardless of whether the block c is in the set of future splitters or not. Below we provide a counterexample in which states that belong to the same block in the final partition are not probabilistic bisimilar. Consider the labelled Markov chain in Figure 6.19. We generate the initial partition as shown in Figure 6.20 by placing states with the same set of labels in the same block.

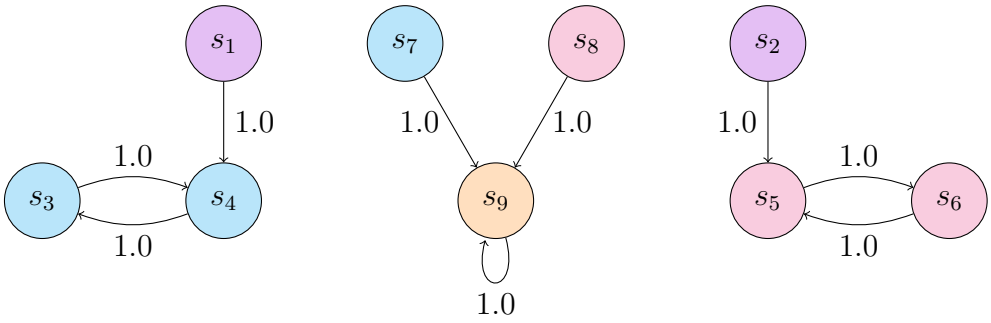


Figure 6.19: A labelled Markov chain.

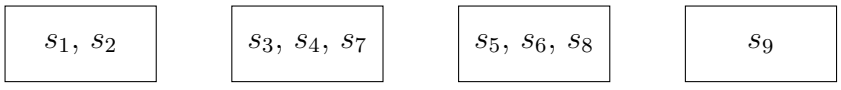


Figure 6.20: The initial partition.

Assume that the last equivalence class is chosen as the splitter X. The state s_9

has two predecessors for which we compute the following.

1 $s_7.sum = 1.0$
2 $s_8.sum = 1.0$

Figure 6.21 depicts a simplified illustration of the execution of the method split. The blocks $\{s_1, s_2\}$ and $\{s_9\}$ are not displayed, since they do not contain any states that are predecessors of state s_9, therefore, they are not refined. During the refinement, states s_7 and s_8 are placed in the blocks of the roots of their corresponding trees, since the splay trees were empty.

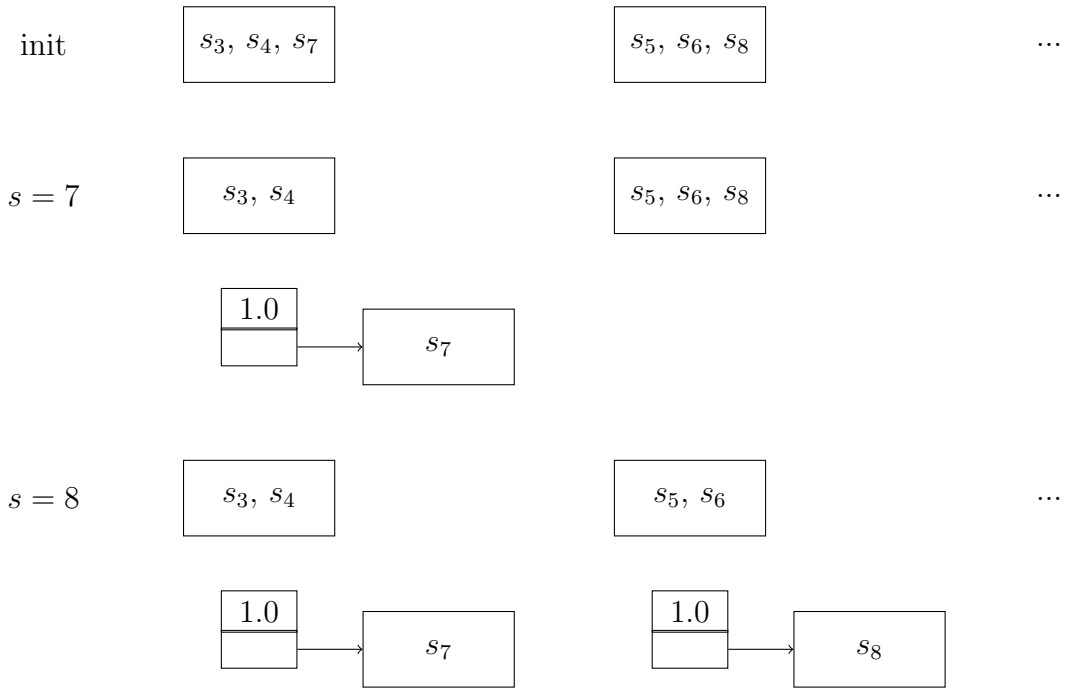


Figure 6.21: The execution of the method split.

According to the paper, we exclude the largest subblock of each original block that contained a predecessor of the splitter X. The largest subblock of $\{s_3, s_4,$

$s_7\}$ is $\{s_3, s_4\}$, while the largest subblock of $\{s_5, s_6, s_8\}$ is $\{s_5, s_6\}$. Furthermore, the splitter X was not refined and, therefore, cannot be used as a splitter again. Hence, the resulting partition is presented in Figure 6.22. The three blocks that are in the set of future splitters, namely $\{s_1, s_2\}$, $\{s_7\}$ and $\{s_8\}$ do not have any predecessors, thus there is no further refinement of the partition and the partition shown in Figure 6.22 is also the final partition.

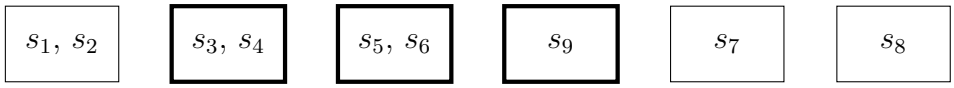


Figure 6.22: The resulting partition.

States s_1 and s_2 are identified as probabilistic bisimilar; however, state s_1 transitions to the second block with probability 1.0 and to the third block with probability 0.0, while state s_2 transitions to the second block with probability 0.0 and to the third block with probability 1.0. According to Definition 3, states s_1 and s_2 are not probabilistic bisimilar.

In the modified algorithm presented in Algorithm 10, we add the blocks $\{s_3, s_4\}$ and $\{s_5, s_6\}$ to the set of future splitters. Using either of these blocks as the splitter results in state s_1 and state s_2 being split into separate blocks. The correct final partition is depicted in Figure 6.23.

Valmari and Franceschinis [VF10] also observe this error and provide a counterexample. However, note that in Figure 6.19 all transitions are labelled with

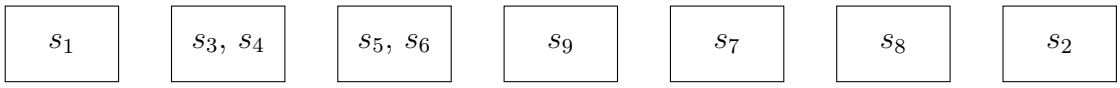


Figure 6.23: The correct final partition.

probability one, thus there is no randomization in our example, in contrast with the one provided in the paper mentioned above.

6.4 Valmari and Franceschinis

6.4.1 The Algorithm

The algorithm developed by Valmari and Franceschinis, presented in Algorithm 11, computes probabilistic bisimilarity for Markov chains [VF10]. The algorithm takes two inputs, namely the transition function τ of the Markov chain and an initial partition of the state space P. As with Derisavi's algorithm, discussed in Section 6.3.1, we use the algorithm to compute probabilistic bisimilarity for labelled Markov chains by providing an initial partition in which all states with the same labels are in the same equivalence class.

In the algorithm, both states and equivalence classes are identified by positive integers. Let us define the variables used in the algorithm. The array *elems* contains all states in S, such that states that belong to the same equivalence class are next to each other. Therefore, each equivalence class has a segment of the array *elems*.

Algorithm 11: VF(τ, P)

Input: $\tau \in S \to \mathcal{D}(S)$ the transition function
$P \in \mathcal{P}(S)$ the initial partition of the state space S

$I \leftarrow$ the set of all blocks in P
foreach $s \in S$ **do** $val[s] \leftarrow 0$
while $I \neq \emptyset$ **do**
    choose an element X from I
    $I \leftarrow I \setminus \{X\}$
    $S_T, B_T \leftarrow \emptyset$
    foreach $t \in X$ **do**
        foreach $s \in$ *predecessors of* t **do**
            if $val[s] = 0$ **then** $S_T \leftarrow S_T \cup \{s\}$
            $val[s] \leftarrow val[s] + \tau(s)(t)$
        end
    end
    foreach $s \in S_T$ **do**
        $c \leftarrow$ the block that contains s
        if c *has no marked states* **then** $B_T \leftarrow B_T \cup \{c\}$
        mark s in c
    end
    foreach $c \in B_T$ **do**
        $c_1 \leftarrow \{s \in c \mid s \text{ is marked}\}$
        $c \leftarrow c \setminus c_1$
        if $c = \emptyset$ **then** give the identity of c to c_1 **else** make c_1 a new block
        $p \leftarrow$ the possible majority candidate of the $val[s]$ for $s \in c_1$
        $c_2 \leftarrow \{s \in c_1 \mid val[s] \neq p\}$
        $c_1 \leftarrow c_1 \setminus c_2$
        if $c_2 = \emptyset$ **then** $n \leftarrow 1$
        else
            sort and partition c_2 according to val, yielding $c_2, \ldots, c_n$
            make each of $c_2, \ldots, c_n$ a new block
        end
        if $c \in I$ **then** $I \leftarrow I \cup \{c_1, \ldots, c_n\}$
        else
            $c_L \leftarrow \max(\{c\} \cup \{c_1, \ldots, c_n\})$
            $I \leftarrow (I \cup \{c\} \cup \{c_1, \ldots, c_n\}) \setminus \{c_L\}$
        end
    end
    foreach $s \in S_T$ **do** $val[s] \leftarrow 0$
end

Each segment is further divided into a first part that contains marked states and a second part that contains unmarked states. For equivalence class c, the array cell $start[c]$ contains the index of *elems* at which c begins (inclusive), the array cell $end[c]$ contains the index at which c ends (exclusive), and the array cell $borderline[c]$ is the index of the first unmarked state in c. For state s, the array cell $location[s]$ denotes the index at which s is located in the array *elems*, while the array cell $block[s]$ denotes the equivalence class to which s belongs. I is the set of blocks which are potential splitters, while X is the current splitter. The set S_T contains states which must be processed, that is, those states which are predecessors to X. Similarly, the set B_T contains the blocks which must be processed, that is, those blocks to which the states in S_T belong. Finally, we use an array *val* of reals, where $val[s]$ is the probability of the state s transitioning to the splitter X.

We refer the reader to the paper [VF10] for a detailed explanation of Algorithm 11. We implemented the algorithm in Java, using the following data structures. The sets I, S_T and B_T as well as the dynamic arrays *start*, *end* and *borderline* are implemented by `ArrayList` with elements of type `Integer`. The array *elems* is represented by `Integer[]` since we sort in line 27 using Java's method `Arrays.sort` with a custom comparator.

6.4.2 An Example

Let us apply the algorithm to our running example, that is, the labelled Markov chain illustrated in Figure 6.1. The initial partition is shown in Figure 6.2. Assume that the second block is picked as the splitter X. We calculate $val[s] = \sum_{t \in X} \tau(s)(t)$, for each s that is a predecessor of some $t \in X$, which corresponds to lines 7–12 in Algorithm 11. Thus, the array $val = [0.5, 0.5, 0.7, 0.0, 1.0, 1.0]$.

In order to visualize the refinement of the partition, we depict the array *elems* as follows. We display the array in segments as it belongs to each equivalence class. States that are marked are coloured grey, while unmarked states are coloured white. The possible majority candidate p, that is, the probability with which possibly a majority of states in an equivalence class transition to the splitter, is displayed above the corresponding equivalence class. Figure 6.24 depicts the changes in the blocks of the partition during the first refinement.

As per lines 13–17, we iterate through those states that are predecessors of the splitter X and mark them. We then iterate through those blocks that contain marked states in lines 18–35. The first block is split into two blocks, namely one containing the marked states $\{s_1, s_2\}$ and one containing the unmarked states $\{s_4\}$. The possible majority candidate p of the new block $\{s_1, s_2\}$, containing the previously marked states, is calculated to be 0.5. Since both states s_1 and s_2 transition to

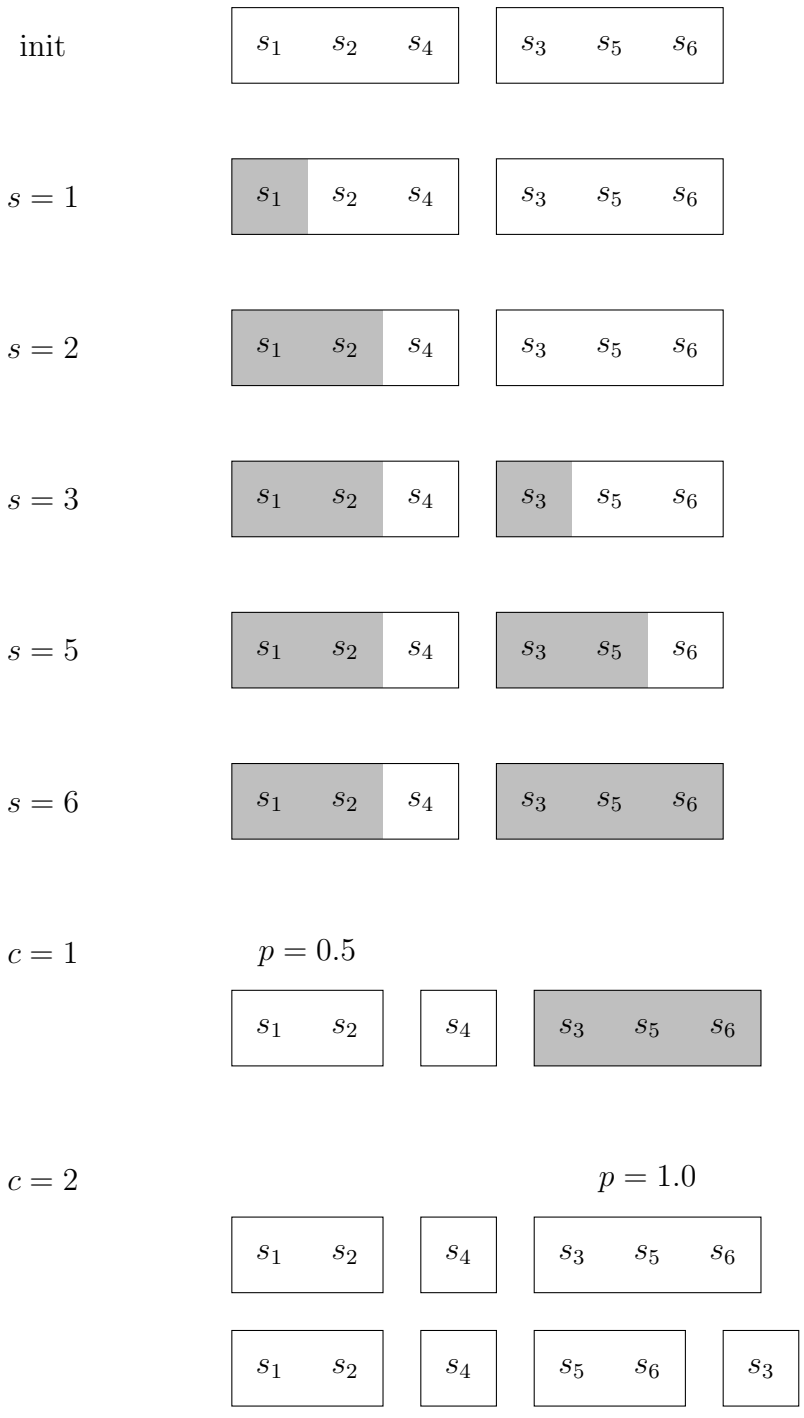


Figure 6.24: The first iteration of the main `while` loop.

the splitter with probability p, that is, $val[1] = val[2] = 0.5 = p$, no further splitting is required. The first equivalence class is in the set I, therefore, all of its subblocks must be added to the set I. We now consider the second original block $\{s_3, s_5, s_6\}$. All states in this block are marked, thus there is no need to split the block and we simply unmark the states. The possible majority candidate is calculated to be 1.0, thus we split the block into two blocks, namely one containing the states that transition to the splitter with probability p, namely $\{s_5, s_5\}$, and one containing the rest of the states $\{s_3\}$. Since the latter block $\{s_3\}$ is a singleton, no further splitting is required. The second equivalence class was used as the splitter X and, thus, is no longer in the set I. Hence, all but one of the subblocks, specifically the largest subblock $\{s_5, s_5\}$, are considered as potential future splitters. The resulting partition is shown in Figure 6.25. Equivalence classes with a thick border are not included in the set I and may not be chosen as a splitter.

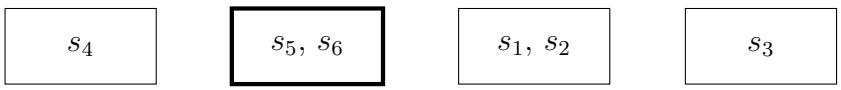


Figure 6.25: The resulting partition.

Assume that the first block is picked as the splitter X. We compute $val[s] = \sum_{t \in X} \tau(s)(t)$, for each of the two predecessors of state s_4. Thus, the array $val = [0.0, 0.5, 0.0, 1.0, 0.0, 0.0]$. Figure 6.26 illustrates the second refinement of the partition.

States s_2 and s_4 are marked. The block $\{s_2, s_1\}$ is split into two blocks, namely

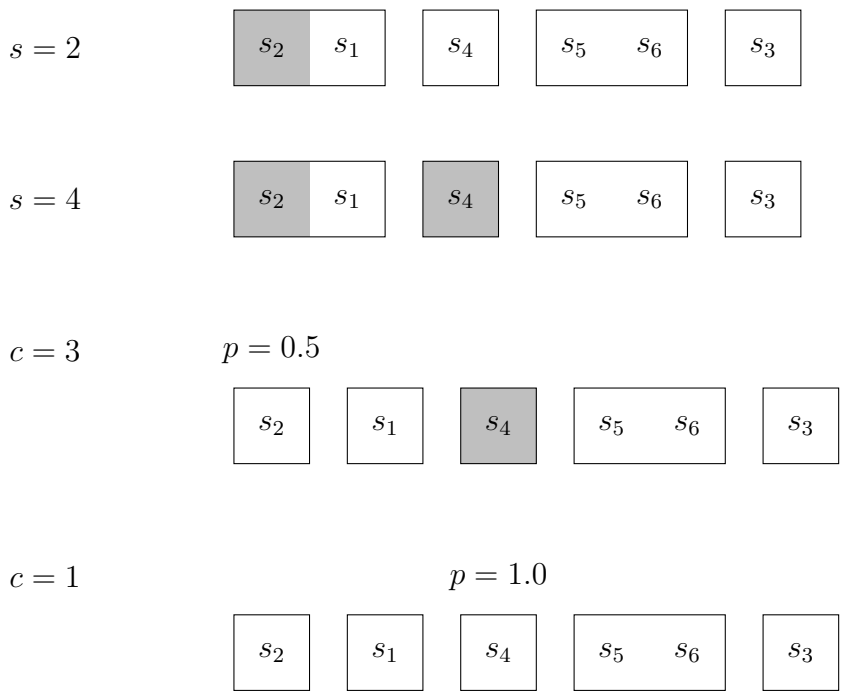


Figure 6.26: The second iteration of the main `while` loop.

one containing the marked states $\{s_2\}$ and one containing the unmarked states $\{s_1\}$. The possible majority candidate of the singleton $\{s_2\}$ is 0.5, since $val[2] = 0.5$. No further splitting is possible. Both subblocks are added to the set I. The block $\{s_4\}$ contains only marked states, thus we simply unmark all of the states in the block. The possible majority candidate of the singleton $\{s_4\}$ is 1.0, since $val[4] = 1.0$. No splitting is possible. Since the splitter X was not refined, it cannot be used as a splitter again. Three more subsequent refinements occur, with the third, fourth and fifth blocks as splitters; however, none of the blocks are split further. Hence, the final partition is shown in Figure 6.27, which matches the results in Sections 6.2.2

and 6.3.2.

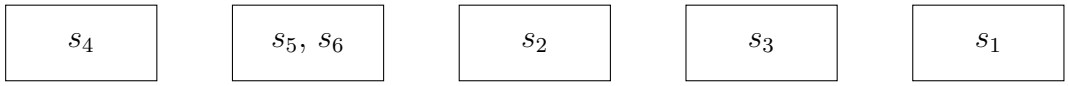


Figure 6.27: The final partition.

6.5 PRISM

6.5.1 Refinement

According to Proposition 6(b), the partition refinement algorithm refines the equivalence relation $\mathcal{R}$ to the equivalence relation

$$\{\, (s,t) \in S \times S \mid \forall C \in S \setminus \mathcal{R} : \tau(s)(C) = \tau(t)(C) \,\}.$$

Hence, given the partition $\mathcal{B}$ corresponding to the equivalence relation $\mathcal{R}$, states s and t remain in the same block if and only if

$$\forall B \in \mathcal{B} : \tau(s)(B) = \tau(t)(B).$$

Recall that for each $s \in S$, $\tau(s)$ is a probability distribution on the set of states S, that is, an element of $S \to [0,1]$. This probability distribution represents all the transitions of the labelled Markov chain the source of which is s. Given a partition $\mathcal{B}$ of S, we can lift a probability distribution μ on states to a probability distribution

on blocks as follows:

$$\mu^{\uparrow}(B) = \sum_{t \in B} \mu(t).$$

Such a probability distribution $\mu^{\uparrow}$ on blocks is an element of $\mathcal{B} \to [0,1]$ and can be represented as a function $\mathbb{N} \to [0,1]$ that maps each block ID to a real number in the interval $[0,1]$. Hence, states s and t remain in the same block if and only if $\tau(s)^{\uparrow} = \tau(t)^{\uparrow}$.

In order to refine a partition $\mathcal{B}$, we compute the lifting $\tau(s)^{\uparrow}$ for each $s \in S$. Each block is split by grouping those states with the same lifting. This can be accomplished by means of the following two functions.

$$\mathit{partition} : \mathbb{N} \to \mathbb{N}$$

$$\mathit{block} : \mathbb{N} \to (\mathbb{N} \to [0,1]) \to \mathbb{N}$$

As we have already seen before in Section 6.1, the function *partition* maps each state ID to the ID of the block to which the state belongs. Given a block B with ID b and a *lifting*, the integer $\mathit{block}[b][\mathit{lifting}]$ is the ID of the block of the refinement to which all states of B with that *lifting* belong.

The following refinement algorithm, Algorithm 12, is similar to the one included in PRISM.[9] This algorithm is based on the one described by Derisavi in [Der07]. For each state, its lifting is computed in line 9–14. Given that lifting, *partition* and

[9]See the class `explicit.Bisimulation` of the PRISM distribution which can be found at the URL github.com/prismmodelchecker/prism.

Algorithm 12: PRISM(τ, *partition*, n)

Input: $\tau \in S \to \mathcal{D}(S)$ the transition function
$partition \in \mathbb{N} \to \mathbb{N}$ the initial partition of the state space S
$n \in \mathbb{N}$ the number of blocks in the partition

```
1  repeat
2      n_old ← n
3      partition_old ← partition
4      n ← 0
5      foreach b ∈ [0, n_old) do
6          block[b] ← ∅
7      end
8      foreach s ∈ S do
9          foreach b ∈ [0, n_old) do
10             lifting[b] ← 0
11         end
12         foreach t ∈ S do
13             lifting[partition_old[t]] ← lifting[partition_old[t]] + τ(s)(t)
14         end
15         if lifting ∉ dom(block[partition_old[s]]) then
16             block[partition_old[s]] ← block[partition_old[s]] ∪ {lifting ↦ n}
17             partition[s] ← n
18             n ← n + 1
19         else
20             partition[s] ← block[partition_old[s]][lifting]
21         end
22     end
23 until n = n_old
```

block are updated in line 15–21.

In the PRISM implementation, *lifting* is represented by the class `Distribution` which has an attribute of type `HashMap<Integer, Double>`. `Distribution`s are considered equal if the relative difference between each corresponding entry is smaller than 10^{-12}. Two data structures are used to represent *block*, namely

`ArrayList<ArrayList<Distribution>>` and `ArrayList<ArrayList<Object>>`.

6.5.2 An Example

Let us apply the algorithm to an example. Consider the labelled Markov chain in Figure 6.28. The initial partition is constructed by placing states labelled with $\{white\}$ in equivalence class 0, states labelled with $\{cyan\}$ in equivalence class 1 and, lastly, states labelled with $\{magenta\}$ in the last equivalence class 2. The initial partition is depicted in Figure 6.29. In PRISM the initial partition is represented by the array [0, 0, 1, 2, 0].

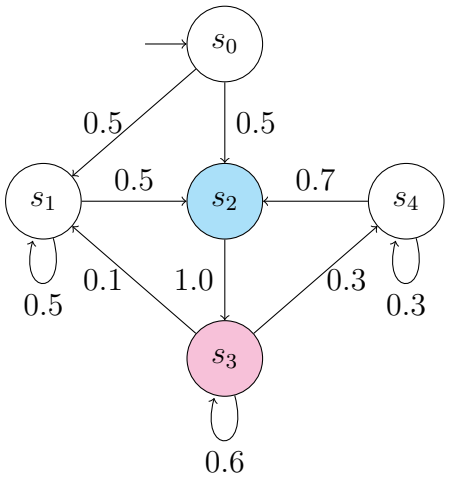


Figure 6.28: A labelled Markov chain.

Figure 6.30 depicts the first refinement. It represents both *block* and *partition*. For each state s, we compute its lifting. This distribution maps each block ID to the probability of s transitioning to that block. We then map this distribution to

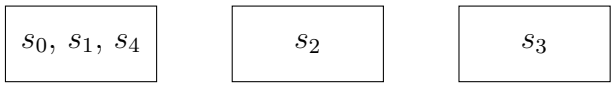


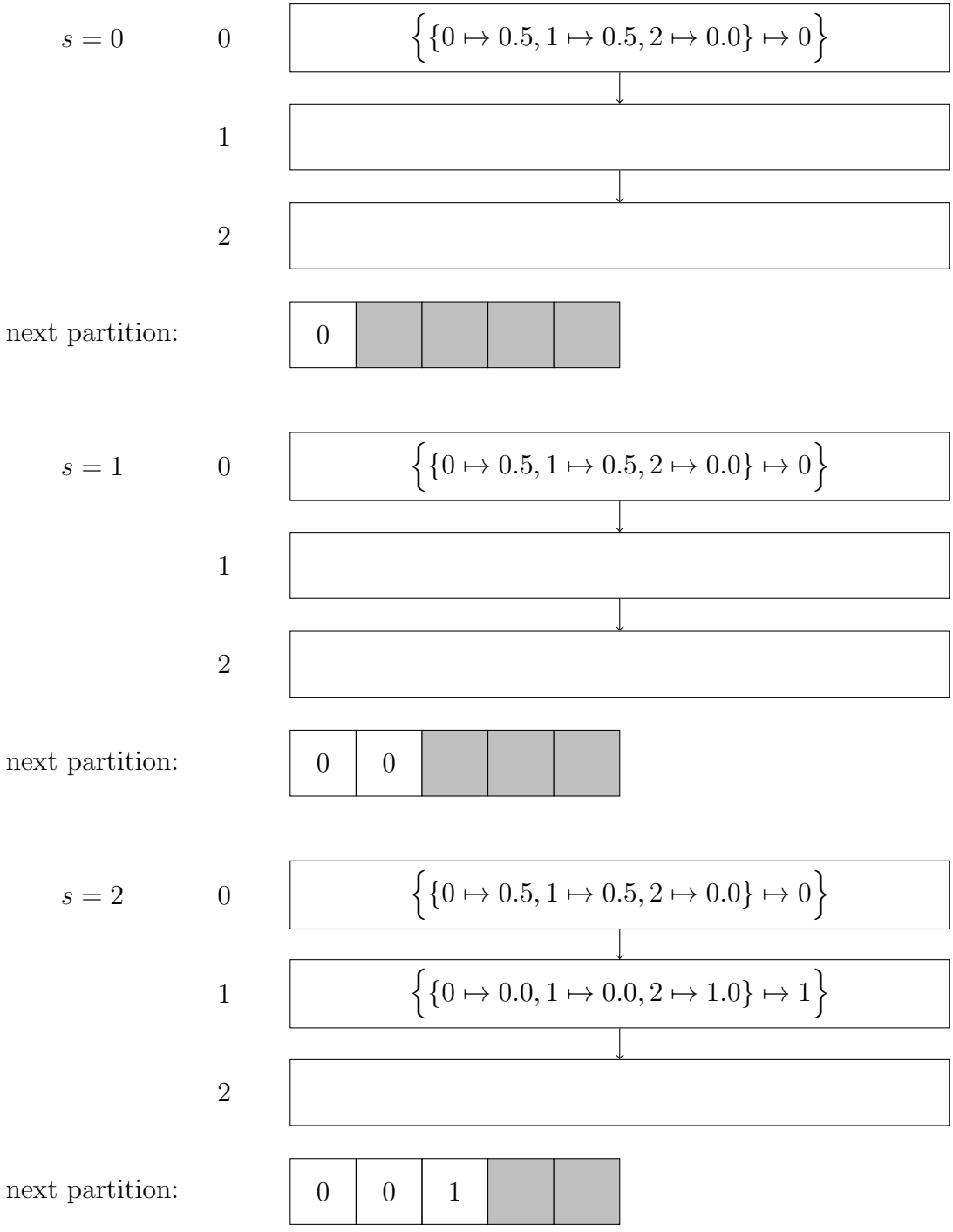
Figure 6.29: The initial partition.

the ID of the block of the refinement. This map is added to *block* at index b, where b is the ID of the block to which s currently belongs. We depict *block* as a list of maps. The list is displayed vertically, indexed by the block IDs on the left. The elements of the list are maps from liftings to block IDs.

We also depict *partition* representing the next partition. Array cells corresponding to states which have not yet been processed are coloured grey.

The lifting for state s_0 is $\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\}$, as s_0 transitions to block 0 with probability 0.5, block 1 with probability 0.5, and block 2 with probability 0.0. Since the refinement is still empty, we map this lifting to its first block, which has index 0, that is, we obtain $\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\} \mapsto 0$. Hence, state s_0 will belong to block 0 in the refined partition. Since state s_0 belongs to block 0 of the current partition, we insert this $\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\} \mapsto 0$ into *block* at index 0.

The lifting for state s_1 is the same as for state s_0, namely $\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\}$. State s_1 also belongs to block 0 of the current partition. Since this lifting is already present at index 0 of *block*, state s_1 will also belong to block 0 of the refined partition. The rest of the states are dealt with in a similar fashion. State s_2's lifting,


$s = 0$
0
$\left\{\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\} \mapsto 0\right\}$
1
2
next partition:
0
$s = 1$
0
$\left\{\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\} \mapsto 0\right\}$
1
2
next partition:
0
0
$s = 2$
0
$\left\{\{0 \mapsto 0.5, 1 \mapsto 0.5, 2 \mapsto 0.0\} \mapsto 0\right\}$
1
$\left\{\{0 \mapsto 0.0, 1 \mapsto 0.0, 2 \mapsto 1.0\} \mapsto 1\right\}$
2
next partition:
0
0
1

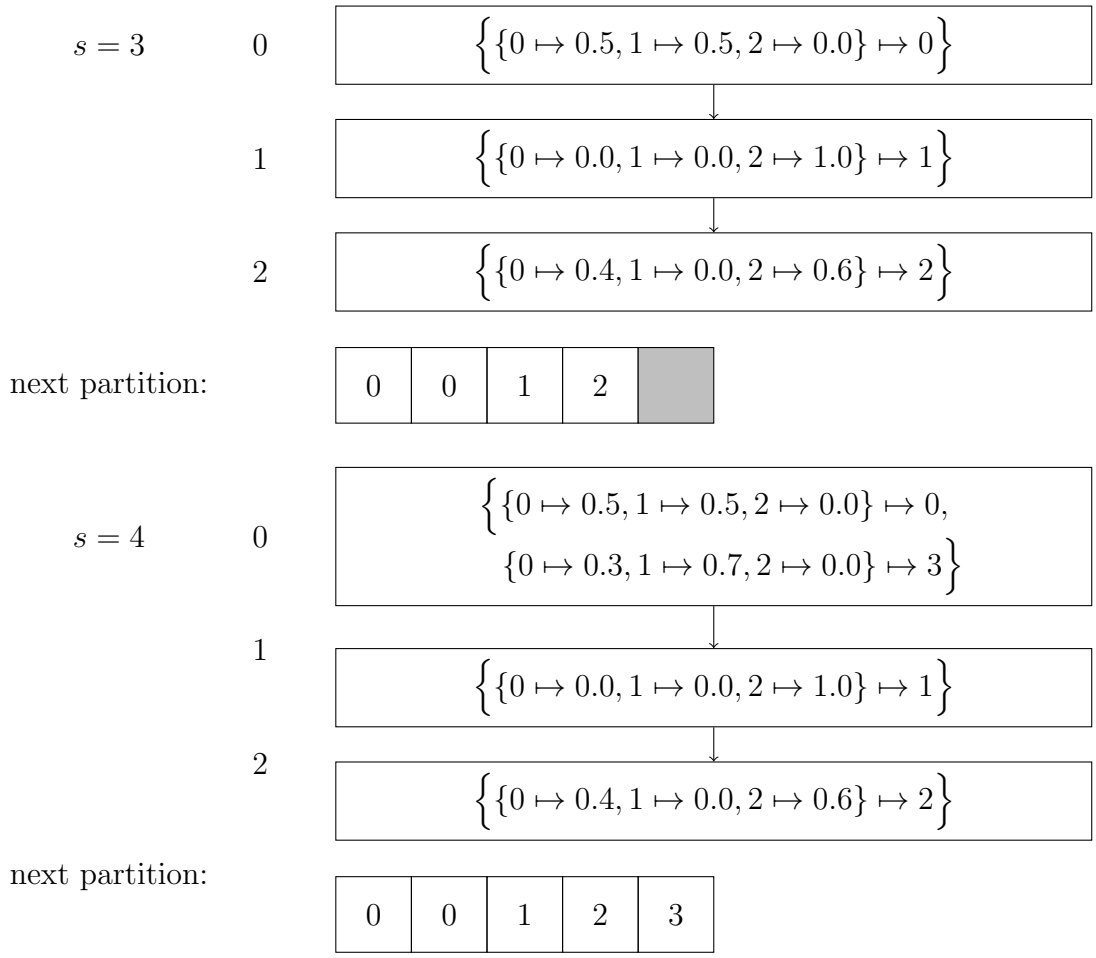


Figure 6.30: The first refinement in PRISM.

namely $\{0 \mapsto 0.0, 1 \mapsto 0.0, 2 \mapsto 1.0\}$ is added to the empty map at index 1 and is assigned 1, thus it is placed into block 1 in the refined partition. State s_3's lifting, namely $\{0 \mapsto 0.4, 1 \mapsto 0.0, 2 \mapsto 0.6\}$ is added to the empty map at index 2 and is assigned 2, thus it is placed into block 2 in the new partition. Finally, state s_4's lifting, namely $\{0 \mapsto 0.3, 1 \mapsto 0.7, 2 \mapsto 0.0\}$ is added to the map at index 0 and is assigned 3, thus it is placed into block 3 in the new partition.

The second refinement step does not modify the partition, thus the algorithm terminates after the second refinement step. The final partition is shown in Figure 6.31.

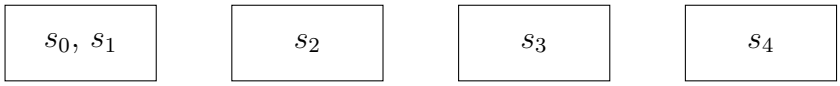


Figure 6.31: The final partition.

6.6 Summary

We adapted Buchholz's algorithm [Buc00] for labelled Markov chains, which can be seen as a special version of a stochastic automaton. We also identified and corrected an error in this algorithm. We presented the algorithm by Derisavi, Hermanns and Sanders [DHS03]. We discussed the error in this algorithm, also observed by Valmari and Franceschinis [VF10], and provided a deterministic counterexample. We then reviewed the algorithm by Valmari and Franceschinis [VF10] and studied a reworked version of the algorithm implemented in PRISM, which is based on the algorithm by Derisavi [Der07]. For each of the four algorithms mentioned above, we described an example of the refinement process and provided some implementation details.

7 Variants of the Algorithms

In Chapter 6 we presented four partition refinement algorithms to compute probabilistic bisimilarity for labelled Markov chains. In this chapter, we discuss the modifications we made to these algorithms, as well as the reasoning behind our data structure choices, if different from those recommended by the authors. We have implemented all of the different variations of the algorithms in Java. Note that for `Double` comparison we use an epsilon of 10^{-12}.

7.1 Buchholz

In addition to the original algorithm, adapted to labelled Markov chains as described in Section 6.2, which we denote as $Buchholz_{original}$, we also implemented the following five variations of Buchholz's algorithm.

Recall that since the sum of the probabilities of the outgoing transitions of a state equal to one, excluding a single block of the partition from the set of splitters does not result in a loss of information. We can also exclude a single subblock

from the set of splitters if the original block has already been used as a splitter, as suggested by Hopcroft [Hop71]. In Buchholz's algorithm [Buc00], all blocks in the initial partition, as well as all subblocks of a block that is split, are considered as future splitters. Thus, we modify the algorithm as follows.

- $Buchholz_{remove\_max}$: If a block c that has been used as a splitter is refined, we exclude the largest subblock of c from the set of future splitters, as done in Derisavi's algorithm [DHS03].

- $Buchholz_{remove\_first}$: If a block c that has been used as a splitter is refined, we exclude the the first subblock of c that is created from the set of future splitters. This is more convenient than excluding the largest subblock, since we do not need to perform any extra computation after each refinement.

- $Buchholz_{remove\_initial}$: If a block c that has been used as a splitter is refined, we exclude the the first subblock of c that is created from the set of future splitters. We also exclude the block containing those states labelled with $\emptyset$ in the initial partition from the set of splitters.

In Section 6.2.1, we mentioned that we implement the dynamic array *class* using an `ArrayList` with elements of type `HashSet` that contain state indices. Similarly, the set I is implemented with `HashSet` and contains block indices. Since both state and block indices are `int`s, the auto-boxing and unboxing of these primitive `int`s,

when adding and removing them from a set, accounts for a significant percentage of the execution time of the algorithm. Thus, we re-implemented Java's generic class `HashSet` to use with the primitive type `int`. We make a copy of $Buchholz_{remove\_initial}$ that uses this `HashSet`, referred to as $Buchholz_{primitive}$.

Lastly, we created a version of the original algorithm, $Buchholz_{original}$, in which the sparse matrix is represented by a list of maps, instead of a list of lists, denoted as $Buchholz_{map}$. For each state s, the list contains a map that represents $\tau(s)$. The map contains the key-value pair $(t, \tau(s)(t))$ for those states t which $\tau(s)(t) \neq 0$.

7.2 Derisavi, Hermanns and Sanders

In Section 6.3.1, we briefly mentioned which data structures we used when implementing Derisavi's algorithm. In the table below, we provide more detail on the recommended data structures mentioned in the paper and the data structures used in our implementation.

Variable	Recommended in the paper	Our implementation
τ	an adjacency list, where the transition probabilities are stored as edge weights	none, the transition probabilities are added to the data structure representing *s.predecessors*

Variable	Recommended in the paper	Our implementation
P	a doubly-linked list whose elements are blocks	`LinkedList<Block>`
I	a doubly-linked list whose elements are blocks	`LinkedList<Block>`
L	a doubly-linked list whose elements are states	`HashSet<State>`
B	a doubly-linked list whose elements are blocks	`HashSet<Block>`
s.predecessors	each state s has a linked list for predecessor states	each state has an attribute of type `LinkedHashMap<State, Double>`, whose entries are pairs of predecessor states and their corresponding transition probabilities
s.successors	each state s has a linked list for successor states	none, this variable is not used in the algorithm

Variable	Recommended in the paper	Our implementation
$c.elements$	each block c contains a doubly-linked list whose elements are states	each block has an attribute of type `HashSet<State>`
c_T	each block c has a splay tree c_T for its subblocks	each block has an attribute of type `SplayTree` whose nodes have an element of type `Block`

Table 7.1: Data structures.

We implemented a version of the algorithm $Derisavi_{original}$ using the data structures recommended in the paper. Using a linked list to implement $c.elements$ is inefficient as it causes line 14 of Algorithm 10 to have $\mathcal{O}(n)$ running time. However, with a hash map we have $\mathcal{O}(1)$ expected running time. Thus, besides $Derisavi_{original}$, the other five variants of the algorithm discussed below are implemented using our choice of data structures. Moreover, we exclude the block containing those states labelled with $\emptyset$ in the initial partition from the set of splitters in line 1 of Algorithm 10. We also remove lines 3–5 and reset $s.sum$ to zero after line 15 instead.

The paper [DHS03] states that using splay tree as the underlying data structure when splitting is essential to obtain the $\mathcal{O}(m \log n)$ running time. Katoen et al. [KKZJ07] mention that red-black trees are often faster in practice. However, in a

later paper [KZH+09] they observe that this was not the case in their experiments. Thus, we implement the following three versions of the algorithm, with different data structures for the subblock tree.

- $Derisavi_{splay}$: The implementation of the splay tree is based on the algorithm by Sleater and Tarjan [ST85].
- $Derisavi_{red-black}$: The implementation of the red-black tree is based on the algorithm described in [CLR89, Chapter 14].
- $Derisavi_{map}$: We represent the subblock tree by `TreeMap<Double, Block>`.

In order to avoid the auto-boxing and unboxing of primitive `double`s when accessing entries of $s.successors$, we re-implemented Java's generic class `LinkedList` to contain two elements, namely a `State` and a primitive `double` value. We make a copy of $Derisavi_{map}$ in which $s.successors$ is represented by this modified `LinkedList` instead of a `LinkedHashMap<State, Double>`, referred to as $Derisavi_{primitive}$.

Finally, we create a copy of $Derisavi_{map}$ in which we consider all blocks in the initial partition as potential splitters, referred to as $Derisavi_{initial}$.

7.3 Valmari and Franceschinis

The paper [VF10] suggests the following two different ways of implementing the algorithm.

- *Valmari*$_{arrays}$: The partition information is represented by the six arrays *elems*, *location*, *block*, *start*, *end* and *borderline*, as described in Section 6.4.1.

- *Valmari*$_{objects}$: States and blocks are represented by classes, as in Derisavi's algorithm. States contain a link to the block to which they belong. Blocks contains two lists, namely one for the marked states and one for the unmarked states.

We also implemented a variation of the six arrays method, called *Valmari*$_{primitive}$, in which we made the following changes to the data structures. In Section 6.4.1, we mentioned that we represent the sets I, S_T and B_T as well as the dynamic arrays *start*, *end* and *borderline* by `ArrayList` with elements of type `Integer`. Since both state and block indices are `int`s, the auto-boxing and unboxing of these primitive `int`s, when adding and removing them from a set or array, accounts for a significant percentage of the execution time of the algorithm. Thus, we re-implemented Java's generic class `ArrayList` to use with the primitive type `int`. We also implemented a variation of Java's method `Arrays.sort` to sort the values in the array *elems* based on the values they index in the array *val*. Hence, the array *elems* can be represented by a primitive array, `int[]`.

We tested the variations of each algorithm by generating a number of labelled Markov chains, using the Erdös-Rényi model [ER59] to generate a random graph,

and confirming that all of the algorithms return the same answer. We used a uniform distribution, that is if there are n outgoing transitions from a state s, then each transition has a probability of $\frac{1}{n}$ to be taken. We used random labelling using a small set of labels. We ran more than 1500 examples of sizes 5, 10, 20, 50, 100, 200, 500, 1000, and 2000 each. The following were non-trivial, that is, the labelled Markov chain contained probabilistic bisimilar states.

Amount	Size
781	5
508	10
273	15
219	20
181	30
259	50
208	75
230	100
161	200
141	300
140	750
150	1000
150	2000

Table 7.2: Non-trivial tests.

8 Experiments

In Chapter 7 we presented the algorithms to compute probabilistic bisimilarity for labelled Markov chains. In this chapter, we provide a comparison of the performance of those algorithms, in terms of memory consumption and execution time, through a few experiments.

We conducted the experiments on a machine with 16 GB of RAM, 12 cores and 2.4 GHz CPU frequency. The machine is running the CentOS Linux operating system version 7.8.2003. We use JDK version 1.8.0_202 and, in order to make a fair comparison, we set the minimum heap size to 1 GB and the maximum heap size to 12 GB for each experiment.

During our experiments, we observed that the following three variants of Buchholz's algorithm, $Buchholz_{remove\_max}$, $Buchholz_{remove\_first}$ and $Buchholz_{remove\_initial}$, behave very similarly in terms of memory consumption and execution time. Thus, in order to simplify the comparison, we will only include $Buchholz_{remove\_initial}$. Likewise, the following three variants of Derisavi's algorithm, $Derisavi_{splay}$, $Derisavi_{map}$

and $Derisavi_{red-black}$, behave very similarly, hence, we only include $Derisavi_{splay}$ in our comparison.

The implementations $Buchholz_{map}$ and $Derisavi_{original}$ are considerably slower and consume more memory than the other implementations of their respective algorithms. In addition, all variations of Valmari's algorithm perform significantly more poorly than the other algorithms. Therefore, we do not include the above mentioned variants in our analysis.

8.1 Crowds Protocol

Crowds is an anonymity protocol developed by Reiter and Rubin [RR98] that protects users' anonymity on the web. The protocol organizes users into groups and selects a random path within a group to route each encrypted message. Web servers or even a corrupt group member cannot determine the origin of a request, as the source is equally likely to have been any member of the group.

The Crowds protocol has been implemented in PRISM's input language by Shmatikov [Shm02] and is included in PRISM's collection of case studies.[10] The algorithm has two parameters, namely `CrowdSize`, which represents the number of honest crowd members, and `TotalRuns`, which represents the number of random

[10]See www.prismmodelchecker.org/casestudies/crowds.php for PRISM's case study on the Crowds protocol.

routing paths to analyze.

Assume that we have twenty honest members in a group and we would like to consider six paths selected by the protocol. Using the following command, we generate the labelled Markov chain of the algorithm with PRISM.

```
prism crowds.pm -const CrowdSize=20,TotalRuns=6 -exporttrans
    crowds.tra -exportlabels crowds.lab
```

The resulting labelled Markov chain has 10,633,591 states and 38,261,191 transitions. The labelled Markov chain is represented in two files, namely `crowds.tra` that contains the transitions and their probabilities and `crowds.lab` that contains the labelling of the initial and final states.

We can now run each algorithm to compute probabilistic bisimilarity on this labelled Markov chain. We compare the memory consumption over time per algorithm for a single run in Figure 8.1. The labelled Markov chain is substantially reduced in size to 50 states and 62 transitions.

By varying the arguments provided to the algorithm and computing probabilistic bisimilarity on the resulting labelled Markov chains, we obtain the graph in Figure 8.2. Note that we run each algorithm 60 times per pair of arguments and execute `System.gc()` before each run to minimize the impact of garbage collection. We discard the first 10 runs to account for the time that the Java virtual machine needs to perform just-in-time compilation and optimization. We then calculate the

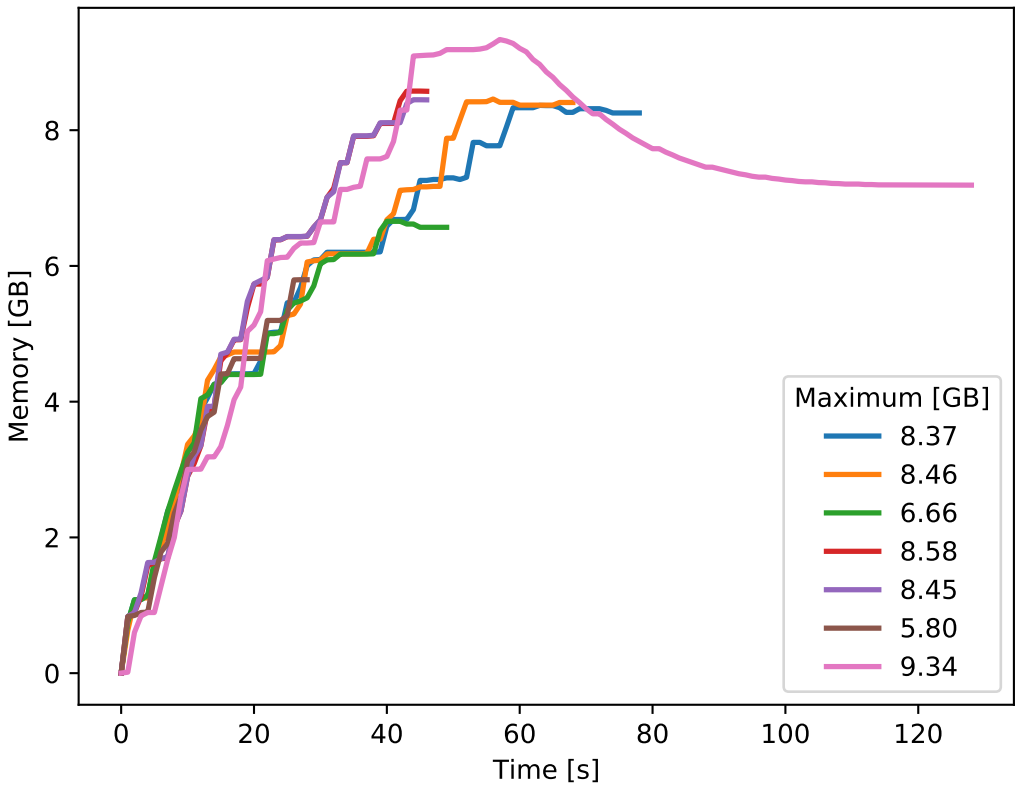


Figure 8.1: This graph depicts the memory used to compute probabilistic bisimilarity on the labelled Markov chain representing the Crowds protocol with twenty honest members for six paths. The colours represent the following algorithms: ● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

average and standard deviation of the execution time of the remaining 50 runs.

In Table 8.1, for each pair of arguments in Figure 8.2, we show the original size of the state space as well as the size of the minimized state space after computing probabilistic bisimilarity.

Notice that in this experiment the variants of the Derisavi algorithm perform the best, followed by the variants of Buchholz algorithm. We also observe that

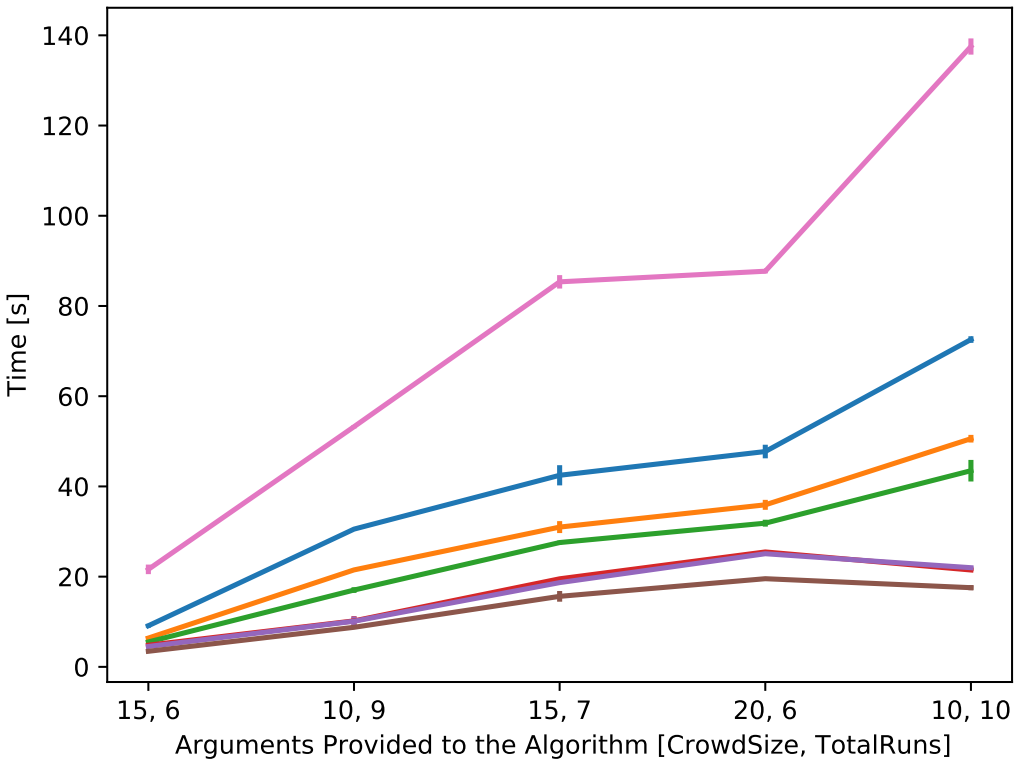


Figure 8.2: This graph depicts the time to compute probabilistic bisimilarity on the underlying labelled Markov chain of the Crowds protocol. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

the variations of the Derisavi algorithm are the most influenced by the number of transitions in the model as seen by the last data point of the graph in Figure 8.2.

8.2 NAND Multiplexing

Von Neumann introduced the redundancy technique called NAND multiplexing to construct reliable computation from unreliable devices [vN56]. When using this

Parameters		Original		Minimized	
CrowdSize	TotalRuns	States	Transitions	States	Transitions
15	6	2,464,167	7,347,928	50	62
10	9	5,971,863	14,285,883	74	92
15	7	8,968,096	26,875,216	58	72
20	6	10,633,591	38,261,191	50	62
10	10	13,201,657	31,677,257	82	102

Table 8.1: The effect of computing probabilistic bisimilarity on the size of the Crowds protocol state space.

technique, a single NAND gate is duplicated N times and N copies of each of the two inputs are made. Each signal from the first input bundle is randomly coupled with a signal from the second input bundle to form the input pair of one of the duplicated NAND gates. The output bundle is fed into the restorative stage to reduce the degradation caused by errors in both the inputs and the faulty devices. To increase efficiency, the restorative stage can be iterated. A critical level ϵ is defined such that if at least $(1 - \epsilon) \times N$ elements of the output set have the same value, the output is decided as that value.

The NAND multiplexing technique has been implemented in PRISM's input language by Norman et al. [NPKS05] and is included in PRISM's collection of

case studies.[11] The algorithm has two parameters, namely `N`, which represents the number of copies of the NAND gate, and `K`, which represents the number of restorative stages.

We generate the labelled Markov chain for the NAND multiplexing unit with fifty NAND gates and ten restorative stages, using the following command.

```
prism nand.pm -const N=50,K=10 -exporttrans nand.tra
   -exportlabels nand.lab
```

The resulting labelled Markov chain has 23,356,802 states and 36,847,227 transitions and is represented by the transition file `nand.tra` and labelling file `nand.lab` that contains the labelling of the initial state. We run each algorithm once on this labelled Markov chain and compare the memory consumption over time in Figure 8.3. The labelled Markov chain is reduced in size to 2 states and 2 transitions.

By varying the number of restorative stages `K`, we construct the graph in Figure 8.4. As described in the previous section, we run each algorithm 60 times per pair of arguments and execute `System.gc()` before each run to minimize the impact of garbage collection. We discard the first 10 runs to account for the time that the Java virtual machine needs to perform just-in-time compilation and optimization. We then calculate the average and standard deviation of the execution time of the remaining 50 runs.

[11] See www.prismmodelchecker.org/casestudies/nand.php for PRISM's case study on NAND multiplexing.

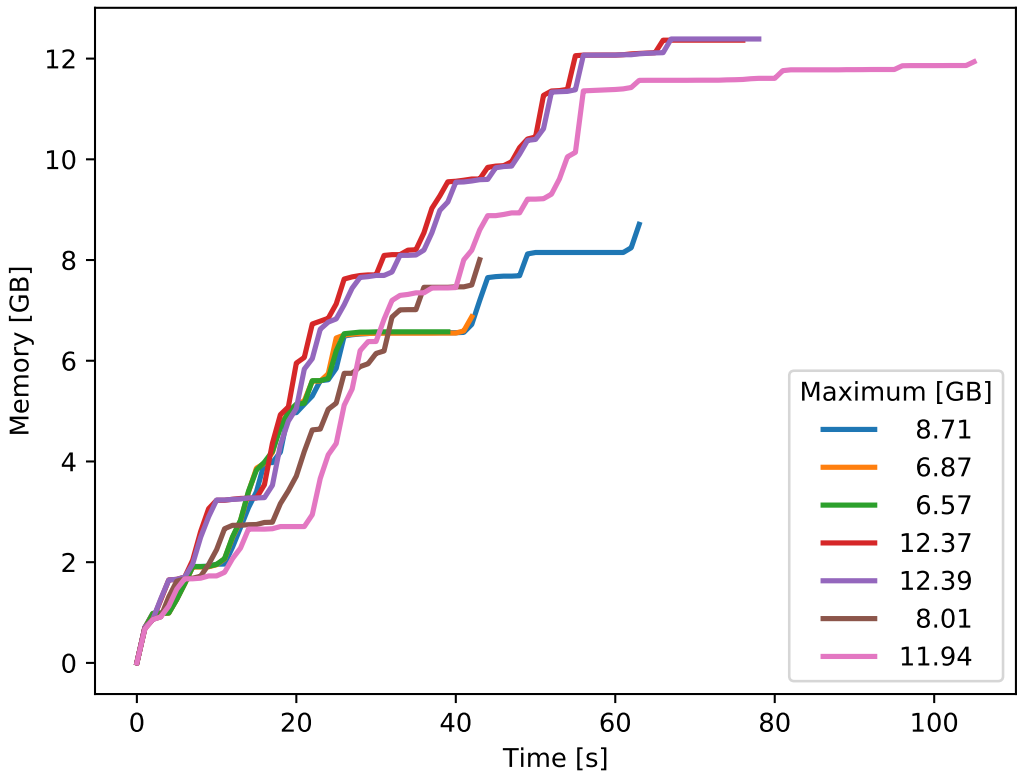


Figure 8.3: This graph depicts the memory used to compute probabilistic bisimilarity on the labelled Markov chain representing the NAND multiplexing technique with fifty NAND gates and ten restorative stages. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

In Figure 8.4, the variants of the Buchholz algorithm are faster than the variants of Derisavi algorithm, contrasting the results of the Crowds protocol in Section 8.1. The non-primitive variants of the Derisavi algorithm as well as the PRISM implementation run out of the allocated 12 GB of memory after ten restorative stages, which corresponds to 23,356,802 states and 36,847,227 transitions. The primitive

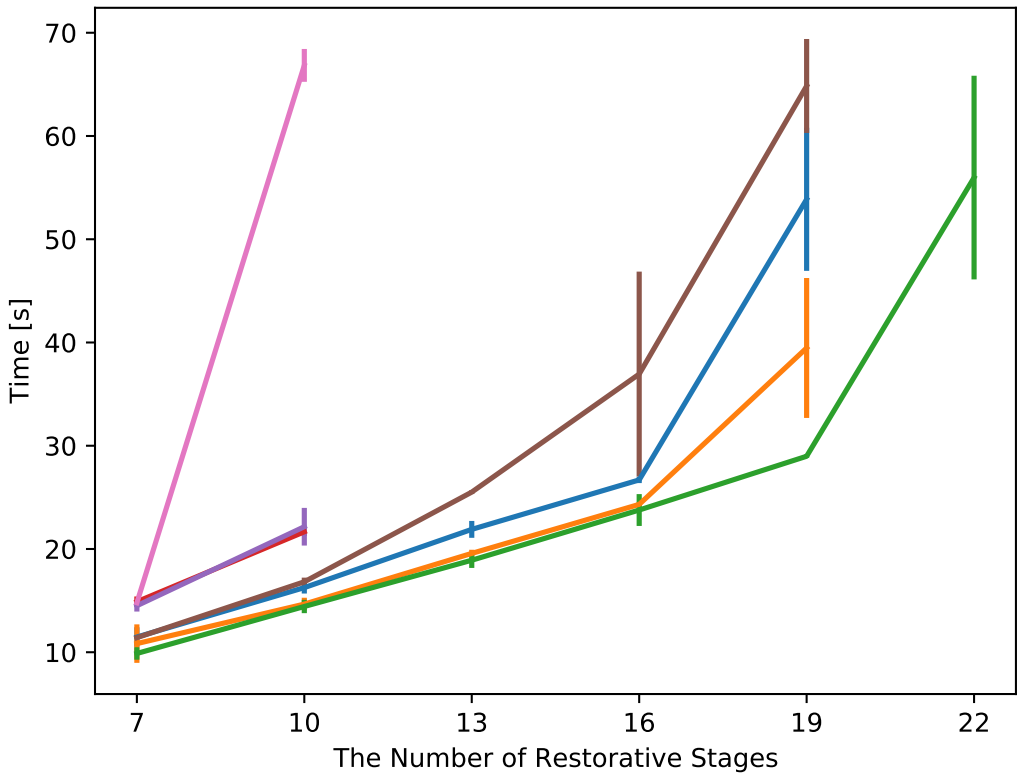


Figure 8.4: This graph depicts the time to compute probabilistic bisimilarity on the underlying labelled Markov chain of the NAND multiplexing technique. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$,
● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

variant of the Derisavi algorithm and the non-primitive variants of the Buchholz algorithm run out of memory after 19 restorative stages, which corresponds to 44,368,652 states and 69,992,427 transitions. This supports the results in Figure 8.3 which demonstrates that the variants of the Buchholz algorithm and $Derisavi_{primitive}$ require much less memory.

In Table 8.2, for each pair of arguments in Figure 8.4, we show the original size

of the state space as well as the size of the minimized state space after computing probabilistic bisimilarity. As indicated by the size of the minimized state space, we can see that in this example, the labelled Markov chain is always reduced to the initial partition, that is, the initial state and one state representing the rest of the state space.

Parameters		Original		Minimized	
N	K	States	Transitions	States	Transitions
50	7	16,352,852	25,798,827	2	2
50	10	23,356,802	36,847,227	2	2
50	13	30,360,752	47,895,627	2	2
50	16	37,364,702	58,944,027	2	2
50	19	44,368,652	69,992,427	2	2
50	22	51,372,602	81,040,827	2	2

Table 8.2: The effect of computing probabilistic bisimilarity on the size of the NAND multiplexing algorithm's state space.

8.3 Tandem Queueing Network

PRISM's implementation of the tandem queueing network is based on the description of an $M/Cox_2/1$-queue in the paper by Hermanns et al. [HMKS99]. The algorithm has one parameter, namely the queue capacity `c`. The model is a continuous-

time Markov chain (CTMC) and is included in PRISM's collection of case studies.[12] Since our algorithms require a discrete-time Markov chain (DTMC) as input, we will need to convert the model.

Assume that we have a queue with a capacity of 150. We first generate the CTMC modelling this queue using PRISM. Our converter `CTMCtoDTMC` then extracts the embedded DTMC from the CTMC by translating the transition rates into transition probabilities as described in Section 6.2. The converter takes in two arguments, that is, the name of the input file containing the CTMC and the name of the output file for the DTMC. This is accomplished by executing the following commands.

```
1 prism tandem.sm -const c=150 -exporttrans tandemCTMC.tra
     -exportlabels tandem.lab
2 java CTMCtoDTMC tandemCTMC.tra tandemDTMC.tra
```

The generated labelled Markov chain has 45,451 states and 157,949 transitions and is represented by the transition file `tandemDTMC.tra` and labelling file `tandem.lab` that contains the labelling of the initial state. We run each algorithm once on this labelled Markov chain and compare the memory consumption over time in Figure 8.5. None of the states in the labelled Markov chain are identified as probabilistic bisimilar, thus the state space is not reduced in size.

By varying the capacity of the queue, `c`, we obtain the graph in Figure 8.6.

[12] See www.prismmodelchecker.org/casestudies/tandem.php for PRISM's case study on the tandem queueing network.

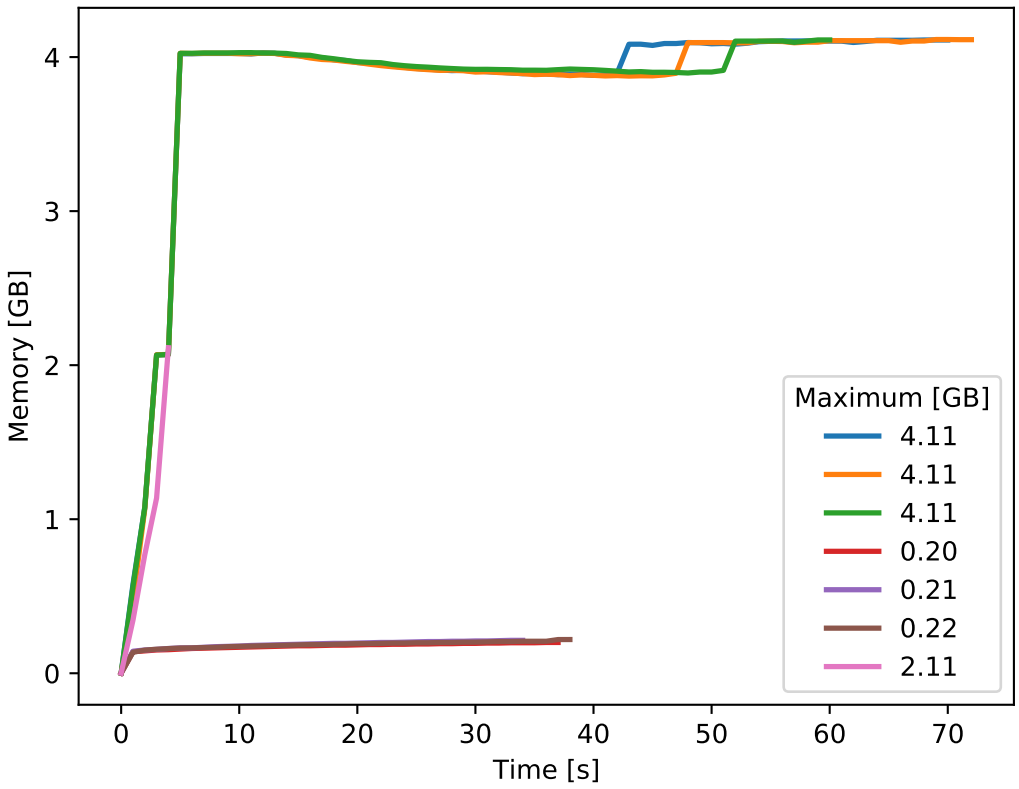


Figure 8.5: This graph depicts the memory used to compute probabilistic bisimilarity on the labelled Markov chain representing the tandem queueing network with a capacity of 150. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

As in the preceding experiments, each algorithm is run 60 times per model with garbage collection performed before each run. The first 10 runs are discarded and we calculate the average and standard deviation of the execution time of the remaining 50 runs.

In Figure 8.6, the variants of the Derisavi algorithm are faster than the variants of the Buchholz algorithm, similar to the results of the Crowds protocol in

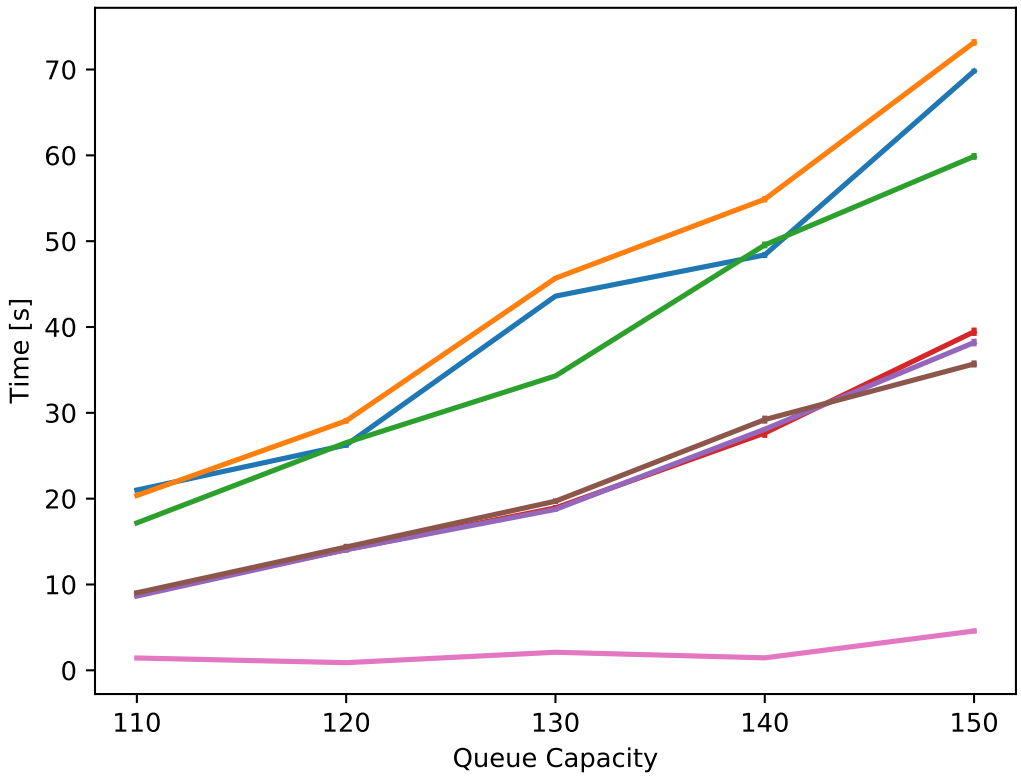


Figure 8.6: This graph depicts the time to compute probabilistic bisimilarity on the underlying labelled Markov chain of the tandem queueing network. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

Section 8.1. However, unlike the previous experiments, PRISM's implementation is the fastest and also consumes approximately 50% less memory than the variants of the Buchholz algorithm as depicted in Figure 8.5. Furthermore, the variants of the Derisavi algorithm require significantly less memory than those of the other algorithms.

In Table 8.3, for each value of the queue capacity in Figure 8.6, we display the

original size of the state space as well as the size of the state space after computing probabilistic bisimilarity. Note that there is no reduction in the state space.

Parameters	Original		Minimized	
c	States	Transitions	States	Transitions
110	24,531	85,029	24,531	85,029
120	29,161	101,159	29,161	101,159
130	34,191	118,689	34,191	118,689
140	39,621	137,619	39,621	137,619
150	45,451	157,949	45,451	157,949

Table 8.3: The effect of computing probabilistic bisimilarity on the size of the tandem queueing network state space.

8.4 Randomized Binary Search

Recall the randomized binary search algorithm from Section 4.5. Assume that we wish to find the element at the 50^{th} position in an array of size 86. We create the following application properties file.

```
target = RandomizedBinarySearch
target.args = 50,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
   ↪ 18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,
   ↪ 36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,
   ↪ 54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,
   ↪ 72,73,74,75,76,77,78,79,80,81,82,83,84,85,86
```

```
3  classpath = <directory containing
       ↪ RandomizedBinarySearch.class>
4
5  @using jpf-label
6  label.class = label.BooleanStaticField
7  label.BooleanStaticField.field =
       ↪ RandomizedBinarySearch.isWorse
8
9  @using jpf-probabilistic
10 listener = probabilistic.listener.StateSpaceText;
       ↪ label.StateLabelText
```

Running JPF with this configuration file gives rise to a labelled Markov chain with 225,907 states and 5,594,758 transitions. This labelled Markov chain is represented by the transition file `RandomizedBinarySearch.tra` and the labelling file `RandomizedBinarySearch.lab` that labels each state with the value of the boolean field `isWorse`, as discussed in Section 4.5. We use our converter, `JPFtoPRISM`, to transform the labelled Markov chain into PRISM's format.

We run each algorithm once on the labelled Markov chain described above and compare the memory consumption over time in Figure 8.7. The state space of the labelled Markov chain is reduced in size to 5,939 states and 281,430 transitions.

By varying the size of the array, `n`, we obtain the graph in Figure 8.8. As in the preceding experiments, each algorithm is run 60 times per model with garbage collection performed before each run. The first 10 runs are discarded and we calculate the average and standard deviation of the execution time of the remaining 50 runs.

The results are very similar to those of the tandem queueing network in Sec-

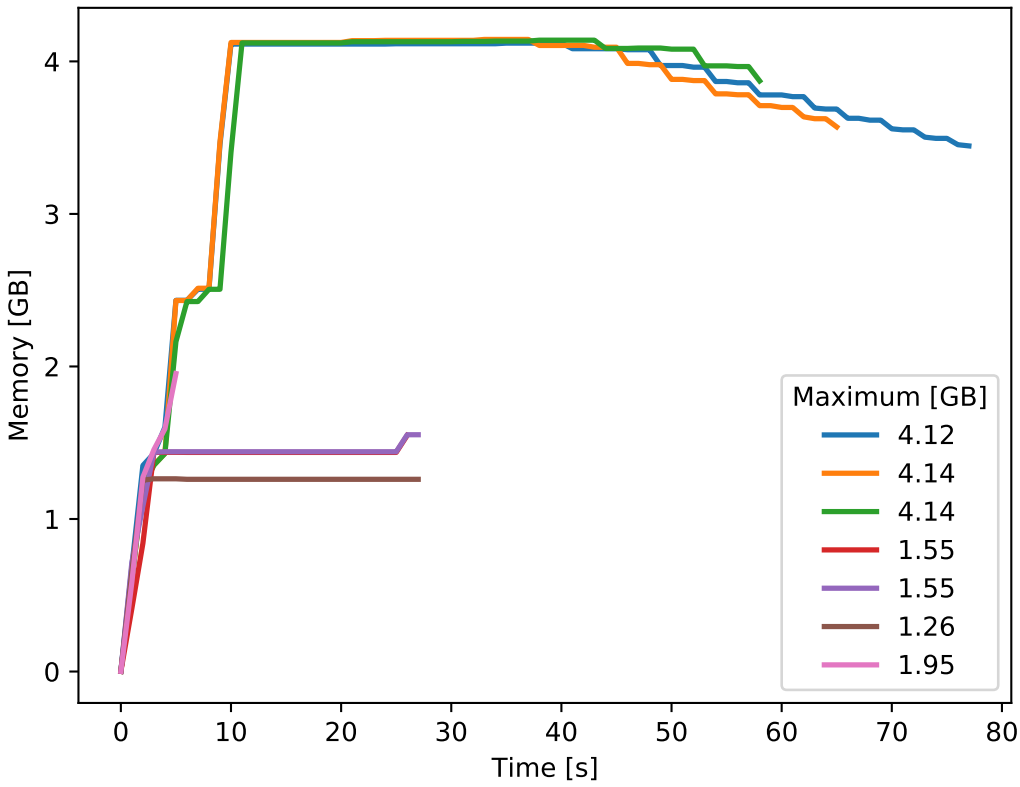


Figure 8.7: This graph depicts the memory used to compute probabilistic bisimilarity on the labelled Markov chain representing the randomized binary search algorithm run with an input array of size 86 and the target value 50. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

tion 8.3. In Figure 8.8, we can see that PRISM's implementation is the fastest and the variants of the Derisavi algorithm are faster than the variants of the Buchholz algorithm. Moreover, the variants of the Buchholz algorithm consume more than double the memory used by the other algorithms, as depicted in Figure 8.7.

In Table 8.4, for each value of the array size in Figure 8.8, we display the

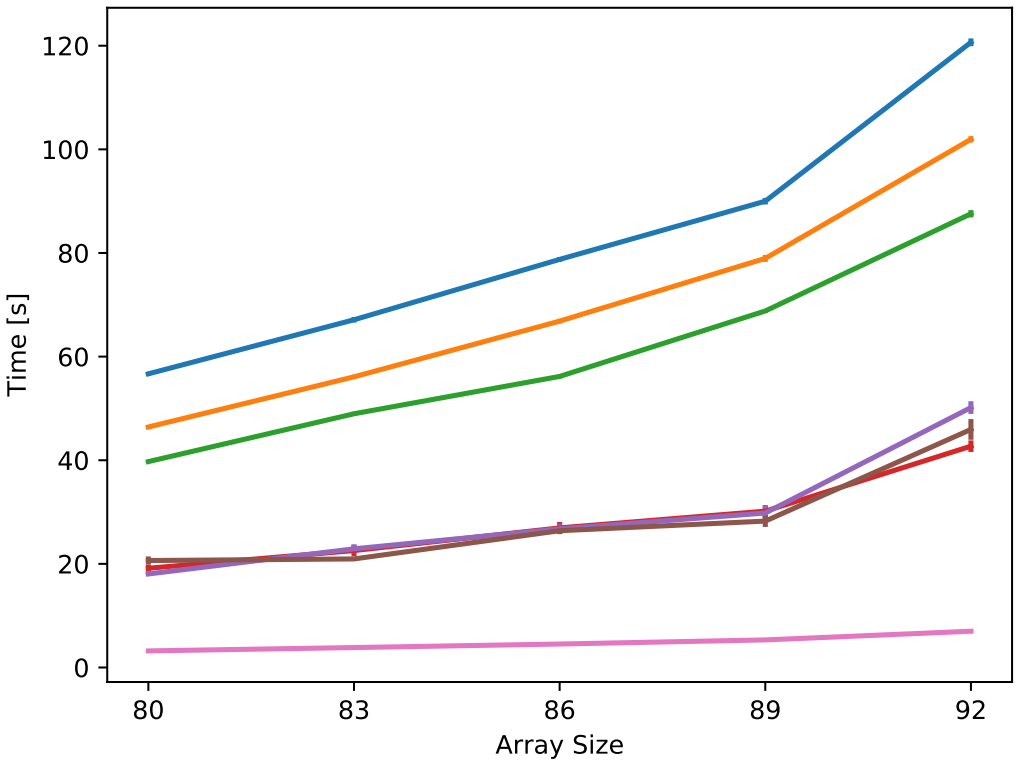


Figure 8.8: This graph depicts the time to compute probabilistic bisimilarity on the underlying labelled Markov chain of the randomized binary search algorithm used to find the 50^{th} element in the input array. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

original size of the state space as well as the size of the state space after computing probabilistic bisimilarity.

Parameters	Original		Minimized	
n	States	Transitions	States	Transitions
80	175,261	4,020,046	5,438	244,150
83	199,909	4,770,193	5,711	263,985
86	225,907	5,594,758	5,939	281,430
89	253,255	6,496,273	6,122	296,106
92	279,734	7,475,268	7,470	366,307

Table 8.4: The effect of computing probabilistic bisimilarity on the size of the state space of the randomized binary search algorithm.

8.5 Summary

When computing probabilistic bisimilarity, if there is a significant reduction in the state space, then the implementations of Derisavi's algorithm perform better compared to Buchholz's algorithm and PRISM's implementation, as seen in the Crowds protocol in Section 8.1.

In Sections 8.2 and 8.3, we present two extremes. The NAND multiplexing example is minimized to the initial partition and, thus, has the smallest possible final partition and the least refinement steps. The labelled Markov chain is also much larger than the other examples presented. In this case, we see that Buchholz's algorithm can handle larger models and is faster than the other algorithms. On the

other hand, in the tandem queueing network example, there is no reduction in the state space and a fair amount of refinement steps. In this case, PRISM performs considerably better than the other algorithms.

The primitive implementations, namely $Buchholz_{primitive}$ and $Derisavi_{primitive}$, are generally faster and consume less memory compared to their corresponding non-primitive variants.

We calculated the average ratio of the running times of the algorithms, using the fastest algorithm as the base time per example. Similarly, we computed the ratio of the memory usage of the algorithms. The results are shown in Figure 8.9. The four experiments are ordered in decreasing size of the state space and also from largest reduction in the state space to the least reduction in the state space.

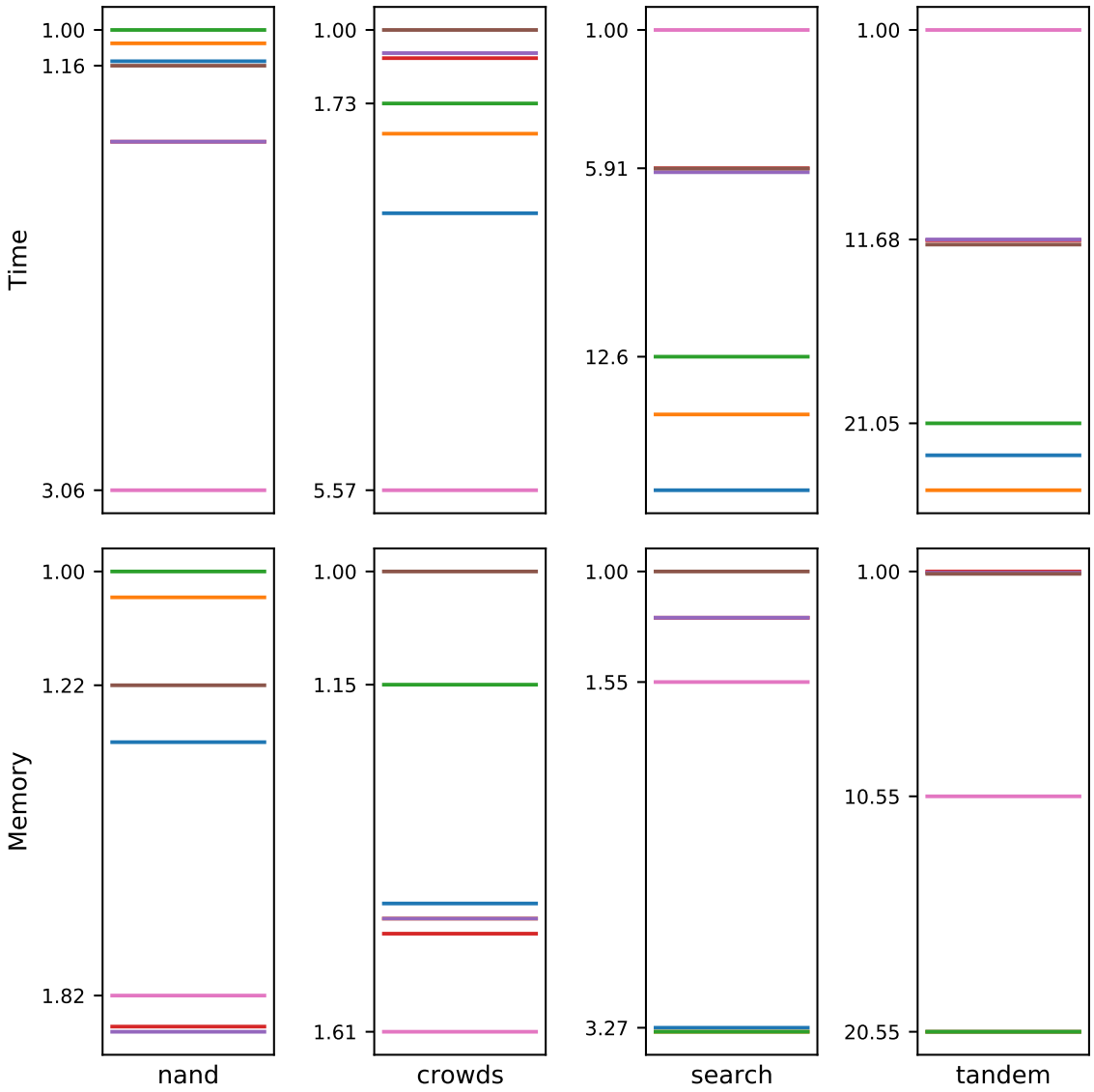


Figure 8.9: This graph summarizes the results of the experiments discussed in this chapter. The colours represent the following algorithms:
● = $Buchholz_{original}$, ● = $Buchholz_{remove\_initial}$, ● = $Buchholz_{primitive}$, ● = $Derisavi_{splay}$, ● = $Derisavi_{initial}$, ● = $Derisavi_{primitive}$, ● = $PRISM$.

9 Conclusion

9.1 Summary

We have developed an extension of JPF, namely jpf-label, which provides users with an easy way to label states with atomic propositions. With the extension we provide twelve different ways to label states, including the labelling of initial and final states, boolean fields and variables, integer fields and variables, method invocations, method returns and the values returned, and thrown exceptions and the exception types. Moreover, our extension can be easily extended to support user-defined labelling functions.

When jpf-label is used in tandem with jpf-probabilistic, the extension of JPF that turns a transition system into a discrete time Markov chain, we can extract the underlying labelled Markov chain of randomized Java code. Our converter transforms the labelled Markov chain generated by JPF into the format of the probabilistic model checker PRISM, allowing us to check properties of the system expressed in logics such as LTL and PCTL. This provides the first model checking

tool that can check probabilistic properties of Java code.

We also investigated the concept of probabilistic bisimilarity for labelled Markov chains, which is a technique used to reduce the state space of a system in order to avoid the state space explosion problem during model checking. We have studied the algorithms to compute probabilistic bisimilarity developed by Buchholz [Buc00], Derisavi, Hermanns and Sanders [DHS03], and Valmari and Franceschinis [VF10]. We implemented these algorithms in Java and introduced a few improvements.

Finally, we compared the performance, in terms of execution time and memory consumption, of the three aforementioned algorithms, as well as the implementation of probabilistic bisimilarity in PRISM [KNP11]. This was done through a series of practical experiments.

9.2 Future Work

jpf-label and jpf-probabilistic, used in conjunction with the probabilistic model checker PRISM, allows us to check properties of randomized Java code. We presented a few examples in Chapter 4 in which we illustrated the functionality of our tool. However, we would like to apply our tool to many more examples and analyze the results.

In Section 2.1.2, we observed that some of the transitions broken by jpf-label need not be broken when the next state is labelled in the same way. This may

occur when the transition is broken by jpf-label and then broken by JPF and the two resulting states have the same set of labels. Avoiding these transitions to be broken by jpf-label would require us to store information about the previous states and remove them if necessary, which would complicate the code of jpf-label significantly. Hence, we leave this as a topic for further research.

Our study of probabilistic bisimilarity for labelled Markov chains introduced many more questions and a lot of work can still be done. We briefly discuss a few suggestions for future research below.

In our implementations of the algorithms to compute probabilistic bisimilarity for labelled Markov chains, we use an epsilon of 10^{-12} for real number comparison, since real arithmetic is not exact. However, when using this method, we could get different results depending on the order in which the states are processed (see Appendix D for detailed examples). Thus, our method of determining equality is reflexive and symmetric, but not transitive. We may want to use rationals instead of reals, as we can compare rationals for exact equality.

Currently, we choose a random splitter from the set of potential splitters during each refinement step. The algorithms could be improved by making a better choice for the splitter. Through experiments, we could determine, for example, whether using larger or smaller blocks as splitters reduces the number of refinement steps.

In the algorithms developed by Buchholz, Derisavi et al. and Valmari et al., the

termination condition is that the set of potential splitters is empty. However, as seen in the examples presented in Chapter 6, multiple unnecessary refinement steps could occur that do not modify the final partition. In order to avoid this we could keep track of a coarser partition of the state space that contains compound blocks. We can then add a second termination condition to the above mentioned algorithms that captures when the current partition is equal to the coarser partition, as is done by Groote, Verduzco and de Vink [GVdV18]. We would also like to adapt the algorithm described in the paper [GVdV18] for labelled Markov chains, implement the algorithm in Java and compare its performance.

We implemented the algorithm by Derisavi, Hermanns and Sanders [DHS03] with three different underlying data structures for the subblock tree, namely a splay tree, a red-black tree and a tree map. We would like to implement this subblock tree using a hash table as well. This is quite challenging, since our current method of deciding equality between real values is not transitive.

We would like to find a more efficient way to do the sorting of the arrays in the algorithm by Valmari and Franceschinis [VF10], which is currently causing the algorithm to have a disadvantage when compared to the other algorithms.

We would like to run many more experiments and perhaps ascertain for which types of labelled Markov chains each algorithm is best suited. Finally, we would also like to determine how the algorithms translate to other models of computation,

such as Markov decision processes.

A Installing jpf-label

We assume that the reader has already installed JPF.[13] To install jpf-label, follow the following steps.

1. Clone jpf-label from github.com/javapathfinder/jpf-label.

2. Build jpf-label with gradle.[14]

3. Add jpf-label to the site.properties file.

[13]Instructions how to install JPF can be found at the URL github.com/javapathfinder/jpf-core/wiki/How-to-install-JPF.

[14]To run the tests and generate the APIs, use gradle with the arguments test and api, respectively.

B PRISM's Keywords

PRISM's input language is a state-based language based on the formalism described in [AH99]. The reserved keywords in the PRISM language are found in Table B.1.

A	bool	C	clock	const
ctmc	double	dtmc	E	endinit
endinvariant	endmodule	endrewards	endsystem	F
false	filter	formula	func	G
global	I	int	invariant	label
max	mdp	min	module	nondeterministic
P	Pmax	Pmin	prob	probabilistic
pta	R	rate	rewards	Rmax
Rmin	S	stochastic	system	true
U	W	X		

Table B.1: The 48 keywords in PRISM.

C Examples Provided by Other Tools

In Table C.1 we provide the number of randomized examples provided with or analyzed by 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	[KNP11]	www.prismmodelchecker.org
QVBS	23 (1)	[HKP+19]	qcomp.org/benchmarks
MRMC	8 (6)	[KZH+11]	www.mrmc-tool.org
PARAM	8 (8)	[HHZ11]	depend.cs.uni-saarland.de/tools/param
PLASMA	6 (1)	[BCLS13]	https://project.inria.fr/plasma-lab
iLTLChecker	5 (5)	[KA04]	osl.cs.illinois.edu/software/iltl
INFAMY	5 (4)	[HHWZ09]	depend.cs.uni-saarland.de/tools/infamy
APMC	4 (3)	[HLMP04]	github.com/ix-labs/apmc
ePMC	4 (4)	[HLS+14]	github.com/ISCAS-PMC/ePMC
IscasMC	4 (4)	[HLS+14]	iscasmc.ios.ac.cn
Storm	4 (1)	[DJKV17]	www.stormchecker.org
CMurphi	3 (1)	[DPIM+04]	bitbucket.org/mclab/cmurphi
PVeSta	3 (1)	[AM11]	maude.cs.uiuc.edu/tools/pvesta
Modest	2 (0)	[HH14]	www.modestchecker.net

Table C.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.

D Order Matters

Consider the following labelled Markov chain.

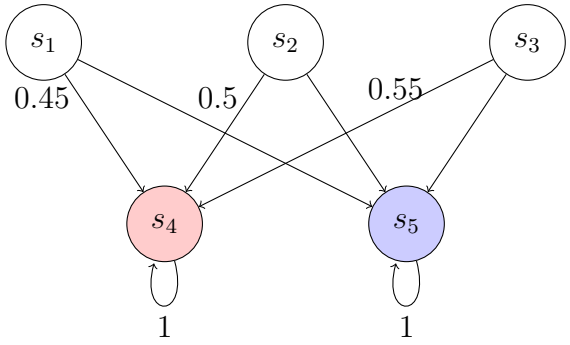


Figure D.1: A labelled Markov chain.

When comparing real numbers for equality, one usually checks whether their absolute difference is smaller than some small real number. For simplicity, let us assume that real numbers are considered equal if their absolute difference is smaller than 0.1.

As we will show below, the order in which the states are considered matters. In particular, if state s_2 is considered before states s_1 and s_3, then there are three probabilistic bisimilarity classes. Otherwise, there are four such classes.

For all algorithms, the initial partition is depicted in Figure D.2. The second and third blocks of the initial partition are singletons and cannot be refined further, thus we focus on the first block for the remainder of this chapter.

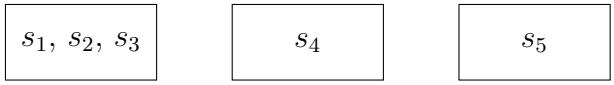


Figure D.2: The initial partition.

D.1 Buchholz

Let us first consider a scenario in which state s_2 is considered before states s_1 and s_3, as shown in Figure D.3.

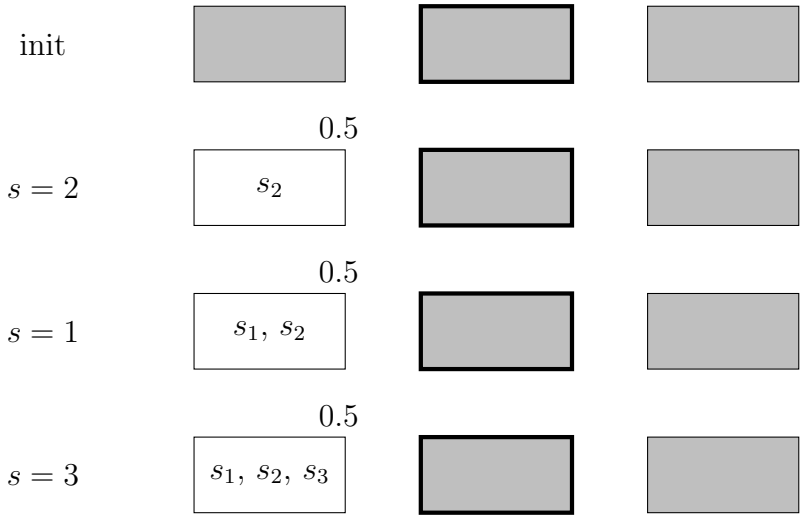


Figure D.3: The first few refinement steps of Buchholz's algorithm when state s_2 is considered before states s_1 and s_3.

In another scenario, state s_1 is considered before state s_2, as shown in Figure D.4.

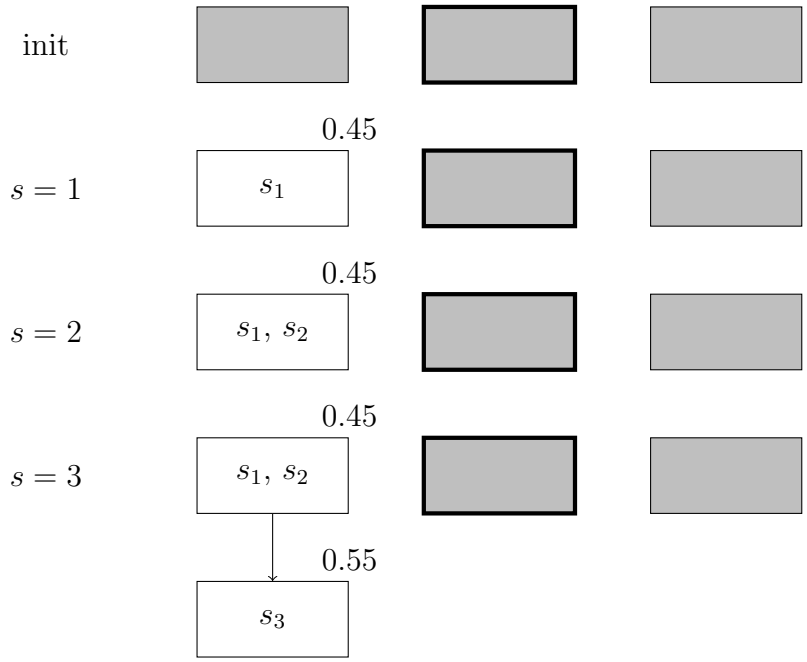


Figure D.4: The first few refinement steps of Buchholz's algorithm when state s_1 is considered before state s_2.

D.2 Derisavi et al.

Let us first consider a scenario in which state s_2 is considered before states s_1 and s_3, as shown in Figure D.5.

In another scenario, state s_1 is considered before state s_2, as shown in Figure D.6.

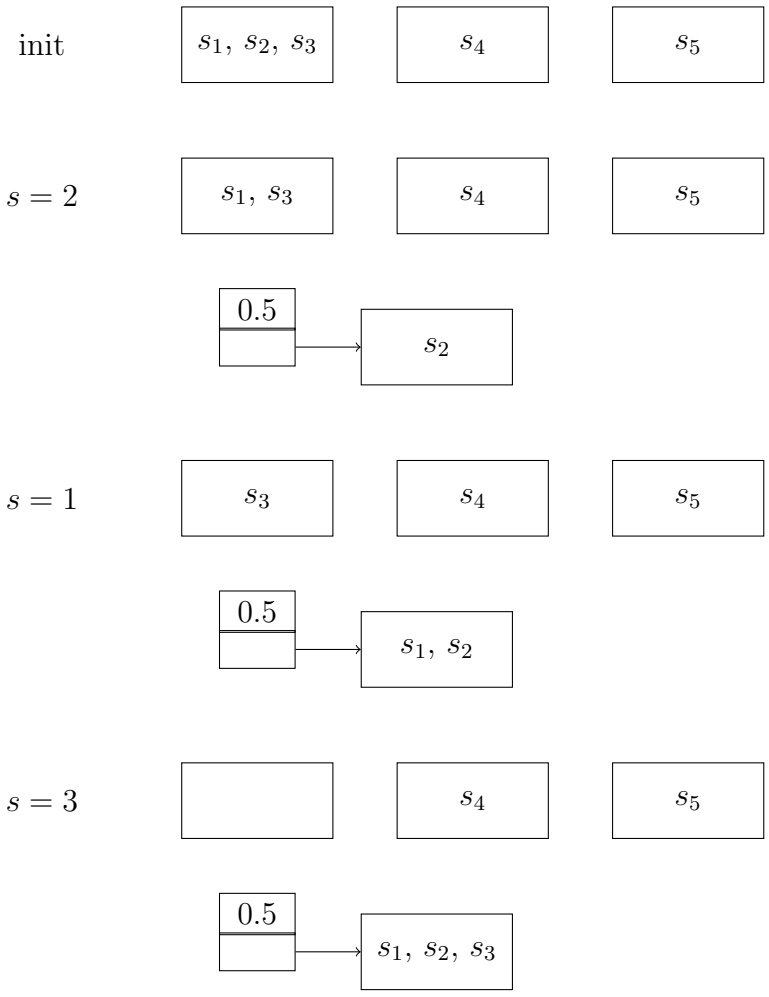


Figure D.5: The first few refinement steps of Derisavi's algorithm when state s_2 is considered before states s_1 and s_3.

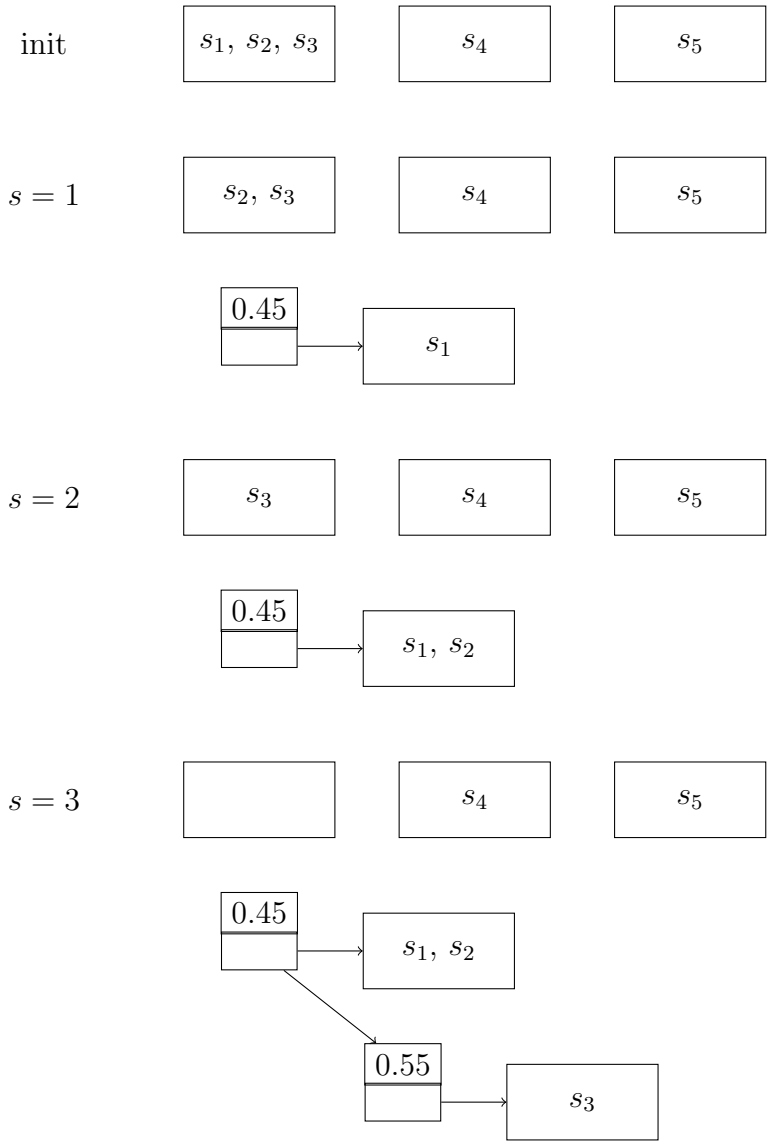


Figure D.6: The first few refinement steps of Derisavi's algorithm when state s_1 is considered before state s_2.

D.3 Valmari et al.

Let us first consider a scenario in which state s_2 is considered before states s_1 and s_3, as shown in Figure D.7.

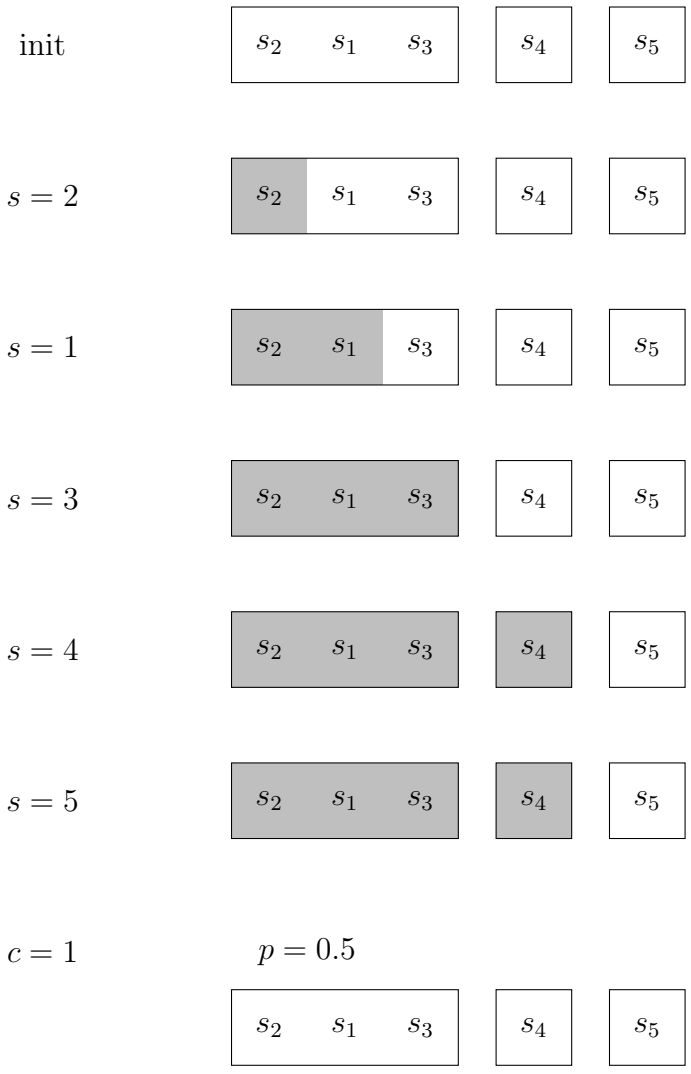


Figure D.7: The first few refinement steps of Valmari's algorithm when state s_2 is considered before states s_1 and s_3.

In another scenario, state s_1 is considered before state s_2, as shown in Figure D.8.

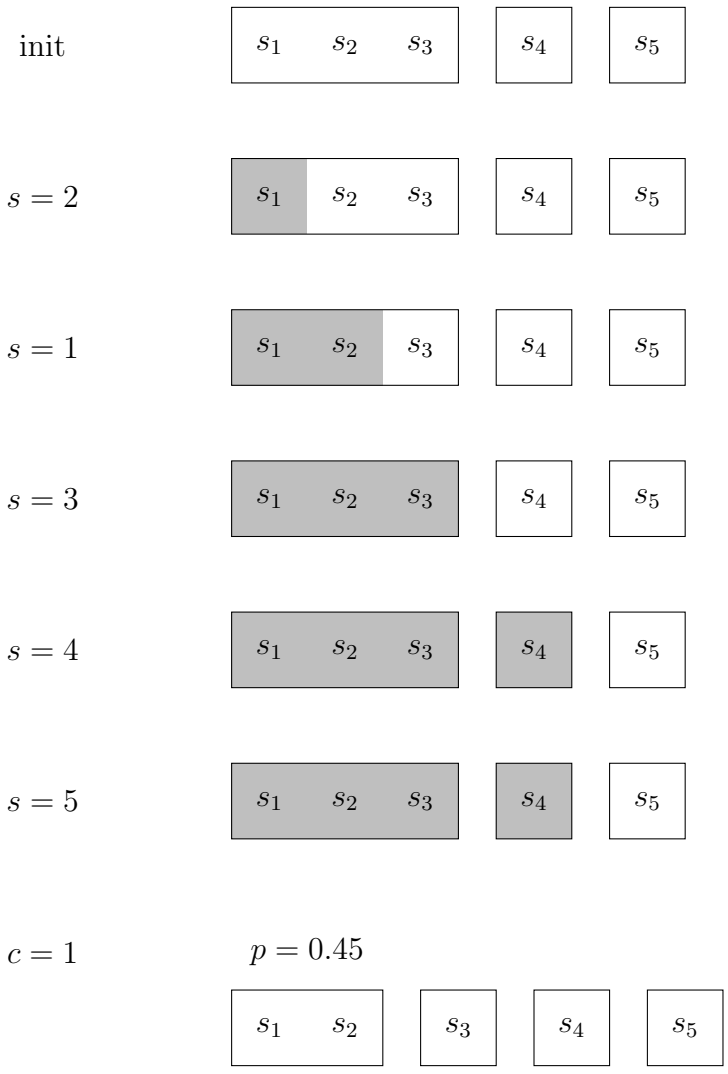


Figure D.8: The first few refinement steps of Valmari's algorithm when state s_1 is considered before state s_2.

D.4 PRISM

Let us first consider a scenario in which state s_2 is considered before states s_1 and s_3, as shown in Figure D.9.

In another scenario, state s_1 is considered before state s_2, as shown in Figure D.10.

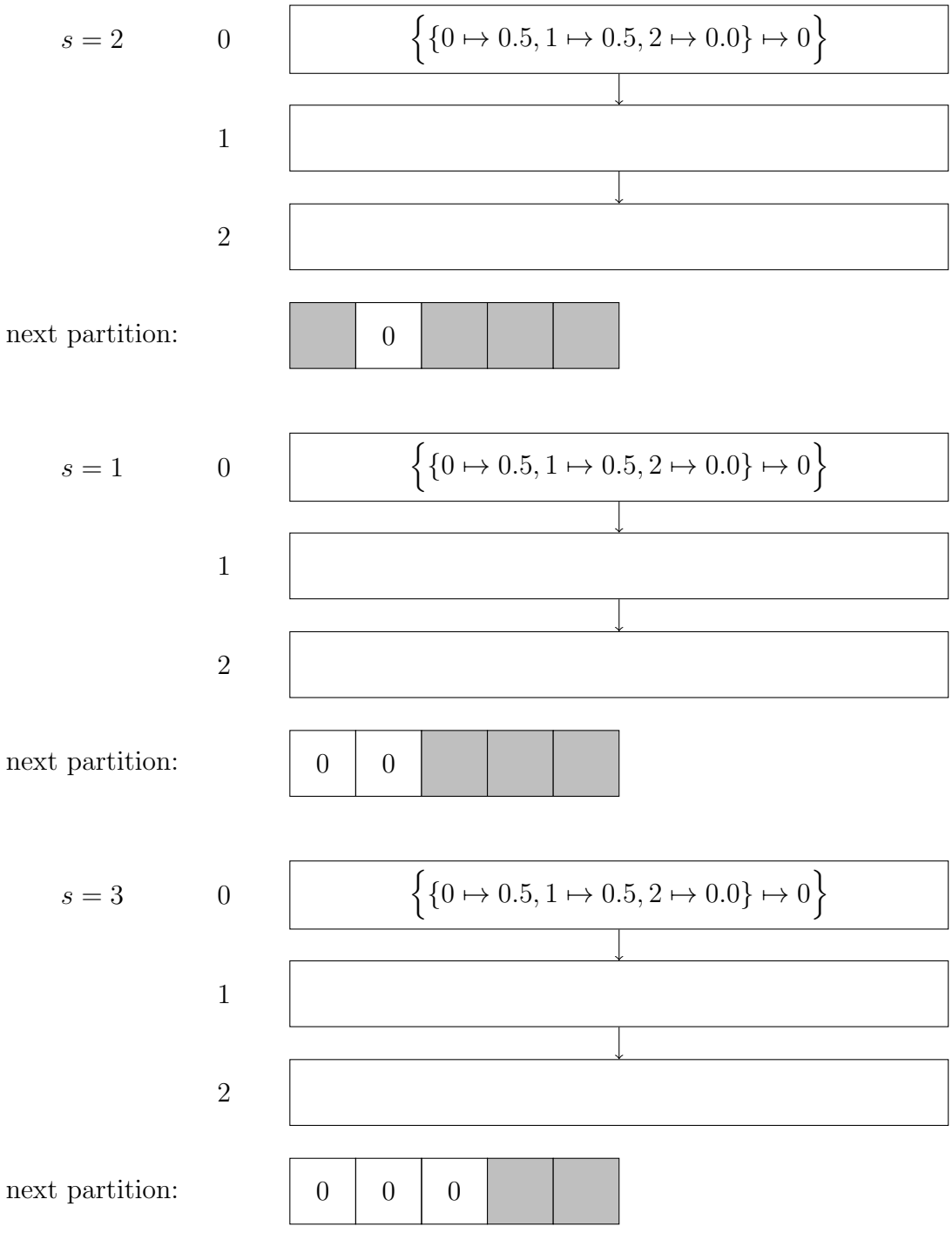


Figure D.9: The first few refinement steps of PRISM's algorithm when state s_2 is considered before states s_1 and s_3.

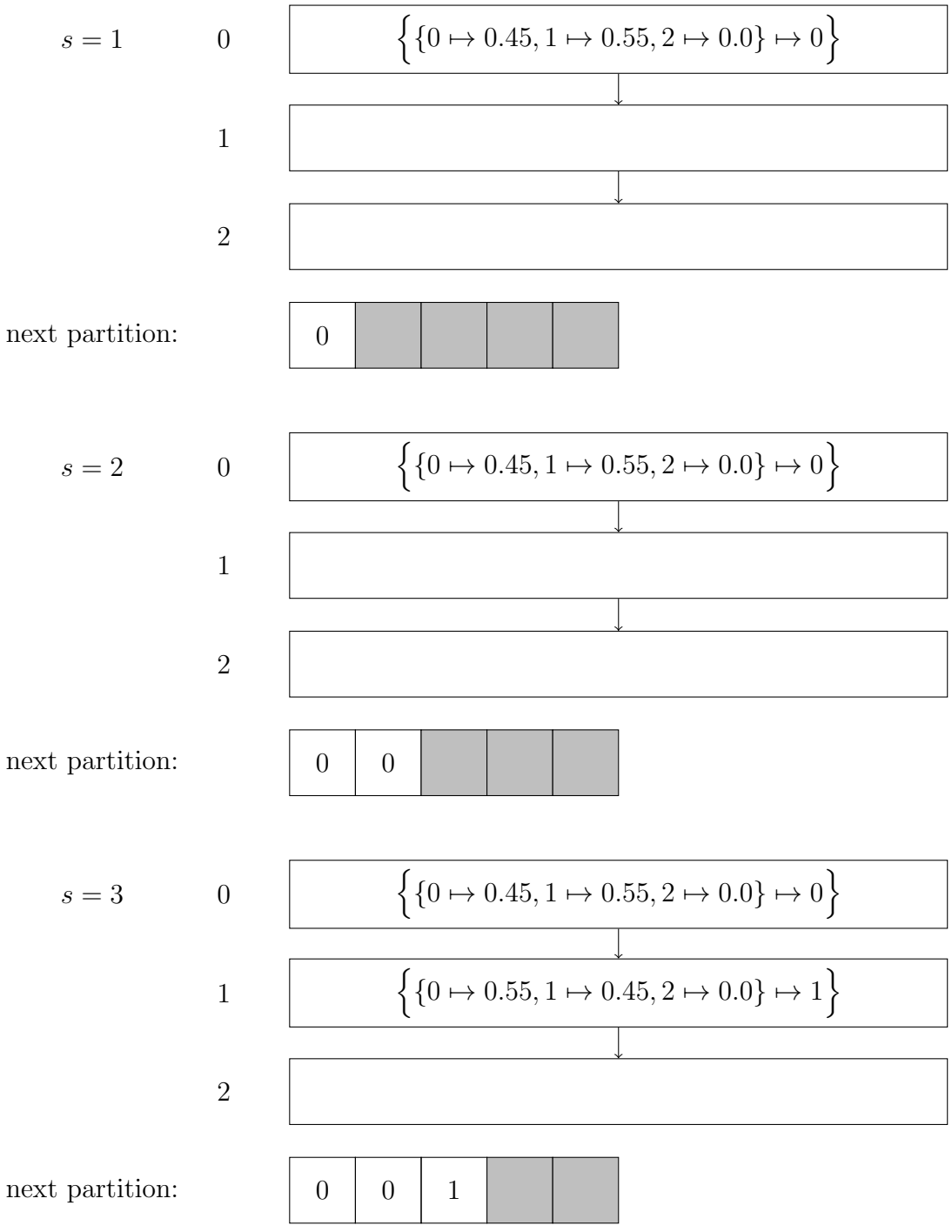


Figure D.10: The first few refinement steps of PRISM's algorithm when state s_1 is considered before state s_2.

Bibliography

[AGR13] 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.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods System Design*, 15(1):7–48, 1999.

[AM11] 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.

[BCLS13] 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.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA, 2008.

[Bot98] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[Buc00] Peter Buchholz. Efficient computation of equivalent and reduced representations for stochastic automata. *International Journal of Computer Systems Science and Engineering*, 15(2):93–103, April 2000.

[CC10] 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.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, USA, May 1981. Springer-Verlag.

[Che19] Khoo Siau Cheng. Personal communication, October 2019.

[Cla08] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2008.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[Der07] Salem Derisavi. Signature-based symbolic algorithm for optimal Markov chain lumping. In *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems*, pages 141–150, Edinburgh, Scotland, UK, September 2007. IEEE Computer Society.

[DHS03] Salem Derisavi, Holger Hermanns, and William Sanders. Optimal state-space lumping of Markov chains. *Information Processing Letters*, 87(6):309–315, September 2003.

[Dij69] Edsger Dijkstra. Structured programming. *Software Engineering Testing*, pages 84–87, 1969.

[DJKV17] 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.

[DP02] Brian Davey and Hilary Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, United Kingdom, 2002.

[DPIM+04] 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.

[ER59] P. Erdös and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FR75] Robert Floyd and Ronald Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, March 1975.

[Gal13] Robert G. Gallager. *Stochastic Processes: Theory for Applications*. Cambridge University Press, 2013.

[Gen06] Rosario Gennaro. Randomness in cryptography. *IEEE Security & Privacy*, 4(2):64–67, 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java language specification*. Oracle, Redwood City, CA, USA, Java SE 8 edition, 2015.

[GL02] 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.

[GVdV18] Jan Friso Groote, Jao Rivera Verduzco, and Erik P. de Vink. An efficient algorithm to determine probabilistic bisimulation. *Algorithms*, 11(9):131, 2018.

[HH14] 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.

[HHWZ09] 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.

[HHZ11] 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.

[HKP+19] 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.

[HLMP04] 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.

[HLS+14] 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.

[HMKS99] Holger Hermanns, Joachim Meyer-Kayser, and Markus Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In Brigitte Plateau, William J. Stewart, and Manuel Silva, editors, *Proceedings of the 3rd International Workshop on Numerical Solution of Markov Chains*, pages 188–207, Zaragoza, Spain, 1999. Prensas Universitarias de Zaragoza.

[Hoa61] Sir Charles Antony Richard Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[Hol04] Gerard Holzmann. *The SPIN model checker*. Addison-Wesley, Boston, MA, USA, 2004.

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189 – 196. Academic Press, 1971.

[KA04] 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.

[KI02] Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Electronic Colloquium on Computational Complexity*, (55), 2002.

[KKZ05] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S. Zapreev. A Markov reward model checker. In *Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems*, pages 243–244, Torino, Italy, September 2005. IEEE Computer Society.

[KKZJ07] Joost-Pieter Katoen, Tim Kemna, Ivan S. Zapreev, and David N. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of

Lecture Notes in Computer Science, pages 87–101, Braga, Portugal, March/April 2007. Springer-Verlag.

[KLHN09] 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.

[KNP11] 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.

[KS60] John G. Kemeny and J. Laurie Snell. *Finite Markov chains*. Springer-Verlag, Heidelberg, Germany, 1960.

[KZH+09] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems*, pages 167–176, Budapest, Hungary, September 2009. IEEE Computer Society.

[KZH+11] 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.

[Lia99] Sheng Liang. *The Java native interface: Programmer's guide and specification*. Addison-Wesley, Reading, MA, USA, 1999.

[LS89] Kim Larsen and Arne Skou. Bisimulation through probabilistic testing. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 344–352, Austin, TX, USA, January 1989. ACM.

[NPKS05] Gethin Norman, David Parker, Marta Z. Kwiatkowska, and Sandeep K. Shukla. Evaluating the reliability of NAND multiplexing with PRISM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1629–1637, October 2005.

[Pnu77] 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.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Torino, Italy, April 1982. Springer-Verlag.

[Rab80] Michael Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, February 1980.

[RR98] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[SB05] 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.

[SB18] Richard Sutton and Andrew Barto. *Reinforcement learning: an introduction*. MIT Press, Cambridge, MA, USA, 2018.

[Sch80] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

[Shm02] Vitaly Shmatikov. Probabilistic analysis of anonymity. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 119–128, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[SZ04] Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2004.

[Tan13] Qiyi Tang. Guiding probabilistic model checkers by reinforcement learning. Master's thesis, University of Oxford, Oxford, UK, September 2013.

[vB17] Franck van Breugel. Probabilistic bisimilarity distances. *ACM SIGLOG News*, 4(4):33–51, October 2017.

[VF10] Antti Valmari and Giuliana Franceschinis. Simple $O(m \log n)$ time Markov chain lumping. In Javier Esparza and Rupak Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 38–52, Paphos, Cyprus, March 2010. Springer-Verlag.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[vN51] John von Neumann. Various techniques used in connection with random digits. In A. S. Householder, George 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, 1951.

[vN56] John von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 43–98. Princeton University Press, 1956.

[XT15] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, June 2015.

[Zha10] Xin Zhang. Measuring progress of model checking randomized code. Master's thesis, York University, Toronto, ON, Canada, July 2010.

[ZvB10] 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.

[ZvB11] 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.