MEASURING PROGRESS OF MODEL CHECKING RANDOMIZED ALGORITHMS

XIN ZHANG

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

MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2010

MEASURING PROGRESS OF MODEL CHECKING RANDOMIZED ALGORITHMS

by **Xin Zhang**

a thesis submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

MEASURING PROGRESS OF MODEL CHECKING RANDOMIZED ALGORITHMS

by **Xin Zhang**

Examination Committee Members:

1. Franck van Breugel
2. Jonathan Ostroff
3. Eric Ruppert
4. Neal Madras

Abstract

Verification of the source code of a probabilistic system by means of an explicit-state model checker is challenging. In most cases, the model checker will either run out of memory or will simply not terminate within any reasonable amount of time. We introduce the notion of a progress measure for such a model checker. The progress measure returns a number in the interval [0, 1]. This number provides us a quantitative measure of the amount of progress the model checker has made. The larger the number, the more progress the model checker has made. We also show how to compute the progress measure for checking invariants. Explicit-state model-checkers usually exploit search strategies such as depth-first search and breadth-first search to explore the transitions. We introduce three new search strategies that take the probabilities associated with the transitions into account. Furthermore, we discuss the implementation of our theoretical framework within the explicit-state model checker Java PathFinder. We compare the amount of progress made by the

different search strategies for eight randomized algorithms implemented in Java.

Acknowledgements

I am extremely thankful to my supervisor Professor Franck van Breugel for his guidance, support, patience and constant encouragement. I thank him for introducing me to the area of model checking and probability theory. I learned many new concepts under his supervision. Not only did he provide me with the support and guidance throughout my research. He also spent so much time on reading my drafts, correcting errors and giving me advice on writing. Without his encouragement, I would never have been able to finish my thesis.

I am grateful to Professor Eric Ruppert, Professor Jonathan Ostroff and Professor Neal Madras for agreeing to serve on my supervisory committee and providing very useful feedback on my thesis. I greatly appreciate Professor Eric Ruppert's efforts in reading my thesis in great depth and providing detailed comments. Many thanks to Professor Jonathan Ostroff for teaching me a lot about software engineering. My heartfelt thanks to Professor Neal Madras his detailed reading of my

thesis and suggesting several improvements. I also thank graduate assistant Ouma Jaipaul-Gill for her help with administrative matters and encouragement. I am thankful to the Computer Science and Engineering Department and NSERC for providing financial support.

I want to thank Trevor Alexander Brown who helped me with some proofs. Our discussions really clarified the concept of model checking.

I thank my fellow graduate students and friends, especially Nastaran Shafiei, Anton Belov, Hui Wang, and Joanna Helga. They did not only give me a lot of new ideas and suggestions, but also provided the wonderful company and cheerful environment in the department.

Table of Contents

1 Introduction

The problem of verifying properties of programs is extremely important, not only for safety critical applications, but also for business applications. During the development of software, more than one third of the time and budget is spent on testing [26], which is meant to find bugs in the software. But despite this effort, particular bugs often remain in the software. On the one hand, for safety critical applications this can cause loss of life. For example, in October 1992 the failure of the London ambulance service computer-aided dispatch [2] was the major reason for 20 to 30 deaths. On the other hand, for business applications the result may be considerable financial losses. One example is the failed Denver airport automated baggage handling system [5]. Started in 1989, the $234 million project was delayed until 1995, costing $500 million more, and in 2005 the system was essentially shut down due to high maintenance costs.

In order to find bugs in software, testing is usually done during and after the development. Since testing can only show the presence of bugs, not their absence

[7], testing does not guarantee the correctness of the system. The main reason why conventional testing is not always sufficient is that it does not usually capture all possible execution paths; or in other words, it is incapable of exploring all the possible states of the program. Verification techniques and tools provide a systematic way to check many or even all possible states of a program. So verification should supplement testing whenever possible.

There are three stages of verification. First, verification should be done during the design stage. After capturing the user requirements, the specification can be developed by using mathematical notation, and tools such as Spin [12] can be used to check for design errors. Second, when the code is being implemented, tools such as Java PathFinder [30] can be used to detect bugs like uncaught exceptions, deadlocks, data races, etc. The last step is to verify the whole system, including user input. Since the users' behaviors are not always predictable, it is useful to exploit a behavior-based tool to verify user input, such as Abbot [8].

Next, we briefly introduce two different approaches to verification.

- Theorem proving. Roughly speaking, a theorem prover is a program that attempts to prove theorems. There are two types of theorem provers, automatic theorem provers and manual theorem provers. The former are fully automatic whereas the latter may need some human input. When applying theorem provers to verification, the properties to be verified are usually mod-

eled as theorems and the program to be verified is usually modeled by axioms and rules. The theorem prover verifies a property of the program by proving the corresponding theorem using the corresponding axioms and rules. For more details about theorem proving, we refer the reader to, for example, [21].

- Model checking. A model checker is a program that systematically explores states of the program being verified. The property being verified is often expressed in terms of a logic. The property determines which states the model checker has to check for a violation of the property. In case a violation is found, the model checker usually provides a counterexample that shows why the program does not satisfy the property. Since the number of states of the program may be very large, in model checking several techniques have been developed to reduce the size of the state space. For more details, we refer the reader to, for example, [1].

In this thesis, we focus on model checking.

In order to check if a program satisfies the property to be verified, an explicit model checker mostly uses a forward strategy to search the state space. The model checker starts in the initial state. When the model checker is in a particular state, it considers the successors of the state. Despite techniques used to reduce the size of the state space, the number of states that can be reached from the initial state

usually remains huge. Moreover, for each state an explicit model checker keeps track of the concrete values of variables. As a consequence, we cannot compress the memory usage for a single state much. In contrast, a symbolic model checker searches a set of states in one single step. Instead of concrete values, a symbolic model checker stores the values of the variables symbolically. In this way, the memory usage is often reduced considerably. Besides the forward search strategy, most symbolic model checkers can also perform backward searches. This strategy identifies those states where the property to be verified is violated, and searches the predecessors of those states. If the initial state is reached, we can conclude that a violation of the property has been found. For a more detailed comparison of explicit and symbolic model checkers, we refer the reader to, for example, [10]. In this thesis, we concentrate on explicit model checking.

Most model checkers consider a model of the system being verified, rather than the source code of the system. A model is usually simpler than the source code and, hence, the model is generally easier to verify. However, the details from which the model abstracts are not considered in the verification effort and, hence, the results obtained when considering a model may be less accurate. For example, consider the failed launch of Ariane 5 on June 4, 1996 [20]. The software of Ariane 5 migrated from Ariane 4, which has a correct model. However, the conversion from a bad 64-bit floating point to a 16-bit integer caused damage of $370 million. Whereas a tool

that checks properties of a model is usually exploited to find errors in algorithms, a tool that considers the source code is generally used to detect coding errors. considers the source code is generally used to detect coding errors. In this thesis, we focus on the model checking of the code of the system, rather than a model of it.

"A randomized algorithm is one that receives, in addition to its input data, a stream of random bits that it can use for the purpose of making random choices" [16]. Each time the algorithm gets a random bit, it resolves a random choice. Randomized algorithms are used in different areas of computer science. For example, in number theory Rabin [25] used the random choices to generate numbers that are prime with high probability. In geometry, Karp [15] used a randomized algorithm to approximate a solution to the traveling salesman problem. Randomized algorithms have several advantages over deterministic algorithms. First, randomized algorithms are usually easier to implement and understand than deterministic algorithms. Second, a randomized algorithm often needs less running time and memory space. In this thesis, we focus on model checking of the source code of randomized algorithms.

On the one hand, ordinary sequential code gives rise to a single execution path. On the other hand, randomized code usually gives rise to many different execution paths. Consider the following Java snippet.

```
Random random = new Random();
long count = 0;
while (random.nextBoolean())
   count++;
```

The above snippet gives rise to a huge number of different execution paths: more than 2^{64}. Hence, it will come as no surprise that an explicit model checker either will run out of memory or will not complete its verification of the above very simple code snippet within any reasonable amount of time. The same applies for most implementations of randomized algorithms as we shall see in Chapter 9 of this thesis.

Since explicit model checkers generally cannot fully verify implementations of randomized algorithms, it would be interesting to extend such a model checker such that it keeps track of the amount of progress it has made with its verification effort. The main question addressed in this thesis is how to define a measure that captures the amount of progress made by an explicit model checker verifying the code of a randomized algorithm. Simply counting the number of (states or) execution paths that have been checked is not very useful for several reasons. First of all, it may be very difficult or even impossible to determine the total number of potential execution paths. Hence, the number of execution paths that have been checked by the model checker gives us very limited information about the amount of progress that has been made. Secondly, some execution paths are more likely to happen

than others. For example, the nonterminating execution path of the above snippet occurs with probability zero. Checking this execution path amounts to no progress at all.

To capture the progress made by the model checker, instead of counting the number of execution paths, we endow the set of potential execution paths with a σ-algebra and a probability measure. In this way, we obtain a probability space of execution paths. As we shall argue, the amount of progress made depends on the property that the model checker is verifying. We restrict our attention to linear time properties. The measure of the set of execution paths, which are relevant to the property under verification and have been checked, gives us a number in the interval $[0, 1]$. This number provides us a quantitative measure of the amount of progress the model checker has made towards verifying the property. The larger the number, the more progress the model checker has made. As far as we know, we are the first to propose a notion of progress. To formalize this notion of progress, we represent an explicit-state probabilistic model checker as a probabilistic transition system and a sequence of transitions. Our formalization of a notion of progress is a major contribution of this thesis.

We prove that our progress measure is a lowerbound of the measure of the execution paths that satisfy the property (provided that no counterexample to the property has been found yet). For example, assume that the progress towards veri-

fying the linear time property ϕ is 0.9999. Then, the probability that we encounter a violation of ϕ is at most 0.0001. Obviously, this is more informative than the model checker providing the message "out of memory."

We provide a characterization of the progress for checking invariants, an important class of linear time properties. This is another major contribution of this thesis. Based on this characterization, we propose an algorithm to compute the progress for invariants.

The work most closely related to ours is the work by Pavese, Braberman and Uchitel [23].[1] They also aim to provide useful feedback when a model checker runs out of memory. Their notion aims to quantify the degree to which the model checker has explored its entire state space. It captures the likelihood of execution paths reaching states that have not been explored by the model checker.

A model checker can use different search strategies to explore the state space. We define three new search strategies: probability-first search, breadth-first probability-second search and randomized search. All take the probabilities of transitions into account when choosing the next transition. This is one of the contributions of this thesis.

We implement our theoretical framework within JPF, an explicit state model checker. This is another major contribution of this thesis. First, we extend JPF

[1] Our work was carried out independently from theirs. We only became aware of a draft version of [23] while finishing this thesis.

to a probabilistic model checker. Second, we implement the algorithm to compute the progress measure for invariants. Moreover, we implement the three new search strategies in JPF. Finally, we also implement a number of randomized algorithms in Java and use our extension of JPF to model check them. We compute the progress by using the algorithm we mentioned above. At the end of this thesis, we shall show the progress made during the search by both our new search strategies and JPF's original depth-first search and breadth-first search. By comparing the results, we shall conclude that for most of these examples, our new search strategies made progress faster than depth-first search and breadth-first search.

2 Some Probability Theory

In this thesis we shall use some basic concepts and results from probability theory to define a notion of progress made when model checking a randomized algorithm. In particular, we shall use the following definitions and theorems.

First, we introduce the notion of a discrete probability distribution and its support.

Definition 2.1.1. Let Ω be a set. A *discrete probability distribution* on Ω is a function $\mu : \Omega \to [0, \infty]$ such that $\sum_{x \in \Omega} \mu(x) = 1$.

Definition 2.1.2. Let Ω be a set and $\mu : \Omega \to [0, \infty]$ be a discrete probability distribution on Ω. The *support of* μ is the set $\{x \in \Omega \mid \mu(x) > 0\}$.

For example, let $\Omega = \{1, 2, 3\}$. Then the function $\mu : \Omega \to [0, \infty]$ defined by $\mu(1) = 0, \mu(2) = 0.5,$ and $\mu(3) = 0.5$ is a discrete probability distribution on Ω, and the support of μ is $\{2, 3\}$.

The proposition below gives an important property of the support of a discrete probability distribution.

Proposition 2.1.3. *The support of a discrete probability distribution is countable.*

Proof. For a proof of this proposition we refer the reader to, for example, [29, Proposition 2.1.2]. □

An important concept in probability theory is a σ-algebra, on which we can define a measure. A σ-algebra is a collection of sets satisfying certain properties, and we give the definition of a σ-algebra below.

Definition 2.1.4. Let Ω be a set. A *σ-algebra* Σ on Ω is a collection of subsets of Ω such that

- $\emptyset \in \Sigma$,
- if $A \in \Sigma$, then its complement $A^c \in \Sigma$, and
- if $A_i \in \Sigma$ for all $i \in \mathbb{N}$, then the union $\bigcup_{i \in \mathbb{N}} A_i \in \Sigma$.

For example, if $\Omega = \{1, 2\}$, then the set $\{\emptyset, \{1, 2\}\}$ is a σ-algebra on Ω. And the set $\{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$ is a σ-algebra on Ω as well.

Furthermore, we introduce the concept of a semi-ring, which is also a collection of sets, but with weaker conditions.

Definition 2.1.5. Let Ω be a set. A *semi-ring* Γ on Ω is a collection of subsets of Ω such that

- $\emptyset \in \Gamma$,
- if $B, B' \in \Gamma$ then $B \cap B' \in \Gamma$, and
- if $B, B' \in \Gamma$ then there exist $B_i \in \Gamma$ for all $i \in \mathbb{N}$ which are pairwise disjoint such that $B \backslash B' = \bigcup_{i \in \mathbb{N}} B_i$.

Note that every σ-algebra is a semi-ring. However, the reverse inclusion does not always hold. For example, if $\Omega = \{1, 2\}$, then the set $\{\emptyset, \{1\}\}$ is a semi-ring, but it is not a σ-algebra. Now we can define a measure on a semi-ring.

Definition 2.1.6. Let Ω be a set. Let Γ be a semi-ring on Ω. A *measure* on Γ is function $\nu : \Gamma \to [0, \infty]$ such that

- $\nu(\emptyset) = 0$, and
- if $A_i \in \Gamma$, for $i \in \mathbb{N}$, are pairwise disjoint and $\bigcup_{i \in \mathbb{N}} A_i \in \Gamma$, then $\nu(\bigcup_{i \in \mathbb{N}} A_i) = \sum_{i \in \mathbb{N}} \nu(A_i)$.

For example, let $\Omega = \{1, 2\}$ and $\Gamma = \{\emptyset, \{1\}\}$. Then the function $\nu : \Gamma \to [0, \infty]$ defined by $\nu(\emptyset) = 0$ and $\nu(\{1\}) = 0.5$ is a measure on Γ.

The following two propositions capture some properties of measures: monotonicity and continuity.

Proposition 2.1.7. *Let μ be a measure on a σ-algebra Σ, and sets $A, A' \in \Sigma$. If $A \subseteq A'$, then $\mu(A) \leq \mu(A')$.*

Proof. For a proof of this proposition we refer the reader to, for example, [3, Section 2] □

Proposition 2.1.8. *Let Ω be a set, Σ be a σ-algebra on Ω, and μ be a measure on Σ. If $A_i \in \Sigma$ for all $i \in \mathbb{N}$, $A_0 \supseteq A_1 \supseteq \ldots$, and $\mu(A_0)$ is finite then $\mu(\bigcap_{i\in\mathbb{N}} A_i) = \lim_{i\in\mathbb{N}}(\mu(A_i))$.*

Proof. For a proof of this proposition we refer the reader to, for example, [3, Theorem 2.1]. □

The following extension theorem states that a measure on a semi-ring can be extended to a measure on a σ-algebra which contains the semi-ring. We shall use this theorem to obtain a measure on sets of execution paths of a probabilistic transition system. This measure will play a key role in this thesis.

Theorem 2.1.9. *Let Ω be a set and Γ be a semi-ring on Ω. If ν is a measure on Γ, then there exists a σ-algebra Σ on Ω and a measure μ on Σ such that*

- $\Gamma \subseteq \Sigma$, *and*
- $\nu(A) = \mu(A)$ *for all* $A \in \Gamma$.

Proof. For a proof of this theorem we refer the reader to [31, Chapter 2]. □

Next, we define the notion of a probability measure.

Definition 2.1.10. Let Ω be a set and Σ be a σ-algebra on Ω. A *probability measure* μ on Σ is a measure such that $\mu(\Omega) = 1$.

For example, let $\Omega = \{1, 2\}$ and $\Sigma = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$. Then the function $\mu : \Sigma \to [0, \infty]$ defined by $\mu(\emptyset) = 0$, $\mu(\{1\}) = 0.5$, $\mu(\{2\}) = 0.5$, and $\mu(\{1, 2\}) = 1$ is a probability measure on Σ.

Finally, we define the notion of a measurable space.

Definition 2.1.11. A *measurable space* is a triple $\langle \Omega, \Sigma, \mu \rangle$ consisting of a set Ω, a σ-algebra Σ on Ω, and a measure μ on Σ.

In this thesis, we shall turn the set of execution paths of a probabilistic transition system into a measurable space. This measurable space will form the basis for our definition of progress.

3 A Model of a Probabilistic Model Checker

In this thesis, we mainly focus on extending JPF to a probabilistic model checker. As we discussed in our introduction, due to the state space explosion problem during model checking a randomized algorithm, the model checker either runs out of memory or it takes an unreasonably long time to finish the model checking. As we already argued in our introduction, it is useful to keep track of the amount of the progress made by the model checker.

In this chapter, we introduce a theoretical framework which we shall exploit in the next chapter to define our measure of progress. We model the randomized code that is checked by an explicit-state probabilistic model checker (like our extension of JPF presented in Chapter 6) as a probabilistic transition system. We turn the set of execution paths of a probabilistic transition system into a measurable space. Our measure of progress will be defined in terms of this measure space. Furthermore, we capture the model checking of some randomized code as a sequence of transitions, which are part of the probabilistic transition system modeling the randomized code.

3.1 A Probabilistic Transition System

We represent the randomized code to be verified by an explicit-state probabilistic model checker, such as our extension of JPF, as a probabilistic transition system.

Definition 3.1.1. A *probabilistic transition system* is a tuple $\langle S, T, AP, s_0, \text{source}, \text{target}, \text{prob}, \text{label}\rangle$ consisting of

- a set S of states,
- a set T of transitions,
- a set AP of atomic propositions,
- an initial state s_0,
- a function source : $T \to S$,
- a function target : $T \to S$,
- a function prob : $T \to (0, 1]$, and
- a function label : $S \to 2^{AP}$

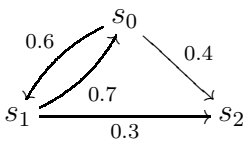
such that

- $s_0 \in S$, and
- for all $s \in S$, $\sum\{\, \text{prob}(t) \mid \text{source}(t) = s \,\} \in \{0, 1\}$.

Instead of $\langle S, T, AP, s_0, \text{source}, \text{target}, \text{prob}, \text{label}\rangle$ we usually write $\mathcal{S}$ and we denote, for example, its set of states by $S_{\mathcal{S}}$.

We shall use the following probabilistic transition system as the running example for the rest of this chapter.

Example 3.1.2. The probabilistic transition system depicted by



has three states and four transitions. In this example, we use the indices of the source and target to name the transitions. For example, the transition from s_0 to s_2 is named t_{02}. Given this naming convention, the functions source and target are defined in the obvious way. For example, $\text{source}(t_{02}) = s_0$ and $\text{target}(t_{02}) = s_2$. The function prob can be easily extracted from the above diagram. For example, $\text{prob}(t_{02}) = 0.4$. The set AP and the function label will not play a role until Section 4.2. There we will discuss them in detail.

For the remainder of this chapter, we fix a probabilistic transition system $\mathcal{S}$. A state is final in $\mathcal{S}$ if it has no outgoing transitions. In Example 3.1.2, the state s_2 is final.

Definition 3.1.3. A state s is *final* in $\mathcal{S}$ if $\sum\{\, \text{prob}_{\mathcal{S}}(t_i) \mid \text{source}_{\mathcal{S}}(t_i) = s \,\} = 0$.

Other than final states, each state has outgoing transitions. In a probabilistic transition system each outgoing transition has an associated probability, and the sum of these probabilities is one. This leads to the fact that the number of outgoing transitions is countable.

Proposition 3.1.4. *Each state has countably many outgoing transitions.*

Proof. This follows immediately from Proposition 2.1.3. □

3.2 A Set of Execution Paths

We classify the potential execution paths of a probabilistic transition system into two categories: infinite execution paths and finite execution paths.

Definition 3.2.1. An *infinite execution path* of a probabilistic transition system $\mathcal{S}$ is an infinite sequence of transitions $t_1 t_2 \ldots$ such that

- for all $i \geq 1$, $t_i \in T_{\mathcal{S}}$,
- $\text{source}_{\mathcal{S}}(t_1) = s_0$, and
- for all $i \geq 1$, $\text{target}_{\mathcal{S}}(t_i) = \text{source}_{\mathcal{S}}(t_{i+1})$.

The set of all infinite execution paths is denoted by $\text{Exec}^{\omega}_{\mathcal{S}}$.

A *finite execution path* of a probabilistic transition system $\mathcal{S}$ is either a finite sequence of transitions $t_1 \ldots t_n$ for some $n \geq 1$ such that

- for all $1 \leq i \leq n$, $t_i \in T_{\mathcal{S}}$,
- $\text{source}_{\mathcal{S}}(t_1) = s_0$,
- $\text{target}_{\mathcal{S}}(t_n)$ is final in $\mathcal{S}$ and
- for all $1 \leq i < n$, $\text{target}_{\mathcal{S}}(t_i) = \text{source}_{\mathcal{S}}(t_{i+1})$,

or the empty sequence ϵ if s_0 is final in $\mathcal{S}$. The set of all finite execution paths is denoted by $\text{Exec}^*_{\mathcal{S}}$.

The set of all execution paths $\text{Exec}_{\mathcal{S}}$ is defined by $\text{Exec}_{\mathcal{S}} = \text{Exec}^{\omega}_{\mathcal{S}} \cup \text{Exec}^*_{\mathcal{S}}$.

For the probabilistic transition system of Example 3.1.2, the sequence $t_{01}t_{10}t_{01}t_{10}\ldots$ is an example of an infinite execution path and the sequence $t_{01}t_{12}$ is an example of a finite execution path.

We use $\text{pref}(\text{Exec}_{\mathcal{S}})$ to denote the set of finite prefixes of execution paths in $\text{Exec}_{\mathcal{S}}$. Note that $\text{Exec}^*_{\mathcal{S}} \subseteq \text{pref}(\text{Exec}_{\mathcal{S}})$. In general, the reverse inclusion does not hold. For example, in the probabilistic transition system of Example 3.1.2, we have that $\epsilon, t_{01}, t_{01}t_{10}, \ldots \in \text{pref}(\text{Exec}_{\mathcal{S}})$, and $\epsilon, t_{01}, t_{01}t_{10}, \ldots \notin \text{Exec}_{\mathcal{S}}$.

In order to identify the last state of an execution path, we extend the function target as follows.

Definition 3.2.2. The function $\text{target}_{\mathcal{S}} : \text{pref}(\text{Exec}_{\mathcal{S}}) \to S$ is defined by

$$\begin{aligned} \text{target}_{\mathcal{S}}(\epsilon) &= s_0 \\ \text{target}_{\mathcal{S}}(t_1 \ldots t_n) &= \text{target}_{\mathcal{S}}(t_n) \end{aligned}$$

To prove that the set pref $(\text{Exec}_{\mathcal{S}})$ is countable, we introduce the following countable sets of states and transitions.

Definition 3.2.3. For each $n \in \mathbb{N}$, the set of states $S^n_{\mathcal{S}}$ and the set of transitions $T^{n+1}_{\mathcal{S}}$ is defined by

$$\begin{aligned} S^0_{\mathcal{S}} &= \{s_0\} \\ S^{n+1}_{\mathcal{S}} &= \{\, \text{target}_{\mathcal{S}}(t) \mid t \in T^n \,\} \\ T^{n+1}_{\mathcal{S}} &= \{t \in T \mid \text{source}_{\mathcal{S}}(t) \in S^n \,\} \end{aligned}$$

Proposition 3.2.4. *For all $n \in \mathbb{N}$, the sets $S^n_{\mathcal{S}}$ and $T^{n+1}_{\mathcal{S}}$ are countable.*

Proof. By induction on n.

- Obviously, $S^0_{\mathcal{S}}$ is countable.
- By the induction hypothesis, the set $T^n_{\mathcal{S}}$ is countable. Therefore, the set $S^{n+1}_{\mathcal{S}}$ is countable as well.
- According to Proposition 3.1.4, each state has countably many outgoing transitions. By the induction hypothesis, the set $S^n_{\mathcal{S}}$ is countable. Hence, the set $T^{n+1}_{\mathcal{S}}$ is countable as well.

□

From the countability of the above introduced sets we can derive that the set pref$(\text{Exec}_{\mathcal{S}})$ is countable as well.

Proposition 3.2.5. *The set* $\mathrm{pref}(\mathrm{Exec}_{\mathcal{S}})$ *is countable.*

Proof. We denote the set of prefixes of $\mathrm{Exec}_{\mathcal{S}}$ of length n by $\mathrm{Exec}^n_{\mathcal{S}}$. Because $\mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) = \bigcup_{n\in\mathbb{N}} \mathrm{Exec}^n_{\mathcal{S}}$ and a countable union of countable sets is countable, it suffices to prove that the set $\mathrm{Exec}^n_{\mathcal{S}}$ is countable for each $n \in \mathbb{N}$. The latter fact we prove by induction on n.

Obviously

$$\mathrm{Exec}^0_{\mathcal{S}} = \{\epsilon\}$$

is a countable set. Assume that the set $\mathrm{Exec}^n_{\mathcal{S}}$ is countable. Since

$$\mathrm{Exec}^{n+1}_{\mathcal{S}} = \{\, et \mid e \in \mathrm{Exec}^n_{\mathcal{S}} \wedge \mathrm{target}_{\mathcal{S}}(e) = \mathrm{source}_{\mathcal{S}}(t) \,\}$$

is a subset of $\{\, et \mid e \in \mathrm{Exec}^n_{\mathcal{S}} \wedge t \in T^{n+1}_{\mathcal{S}} \,\}$ and both $\mathrm{Exec}^n_{\mathcal{S}}$ and $T^{n+1}_{\mathcal{S}}$ are countable, we can conclude that $\mathrm{Exec}^{n+1}_{\mathcal{S}}$ is countable as well. □

Corollary 3.2.6. *The set* $\mathrm{Exec}^*_{\mathcal{S}}$ *is countable.*

We use $|e|$ to denote the length of the execution path e. For example, in the probabilistic transition system of Example 3.1.2, $|t_{01}t_{12}| = 2$.

Furthermore, we use $e[n]$ to denote execution path e truncated at length n. If $|e| \leq n$ then $e[n]$ is equal to e. Otherwise, $e[n]$ consists of the first n transitions of e. Note that $e[n] \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}})$.

In the probabilistic transition system of Example 3.1.2, for the infinite execution

path $e = t_{01}t_{10}t_{01}t_{10}\ldots$, $e[1] = t_{01}$, $e[2] = t_{01}t_{10}$, $e[3] = t_{01}t_{10}t_{01}$, and so on. For the finite execution path $e = t_{01}t_{12}, e[1] = t_{01}$, and $e[2] = e[3] = \cdots = t_{01}t_{12}$.

Let e_1, $e_2 \in \text{pref}(\text{Exec}_{\mathcal{S}})$. We use $e_1 \precsim e_2$ to denote that e_1 is a prefix of e_2. In the probabilistic transition system of Example 3.1.2, $t_{01} \precsim t_{01}t_{10}$, and $t_{01}t_{10} \not\precsim t_{01}t_{12}$.

Furthermore, we use $e_1 \sim e_2$ to denote that e_1 is a prefix of e_2 or e_2 is a prefix of e_1, and we use $e_1 \not\sim e_2$ to denote that e_1 is not a prefix of e_2 and e_2 is not a prefix of e_1. In the probabilistic transition system of Example 3.1.2, $t_{01}t_{10} \sim t_{01}$ and $t_{01}t_{10} \not\sim t_{01}t_{12}$

3.3 A Measurable Space of Execution Paths

As we already discussed in the introduction, we shall define our progress measure by means of a measurable space of execution paths. As we have seen in Definition 2.1.11, a measurable space consists of a set, in this case the set $\text{Exec}_{\mathcal{S}}$ of executions paths, a σ-algebra and a measure. In this section, we introduce a semi-ring and a measure on this semi-ring and use Theorem 2.1.9 to show that there exists a σ-algebra, which extends the semi-ring, and a measure on this σ-algebra, which extends the measure on the semi-ring. The measurable space we define below is similar to the ones studied by Segala [27] and Sokolova, De Vink and Woracek [28].

The elements of the semi-ring are the so-called basic cylinder sets. These are

defined as follows.

Definition 3.3.1. Let $t_1 \dots t_n \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}})$. Its *basic cylinder set* $B^{\mathcal{S}}_{t_1 \dots t_n}$ is defined by

$$B^{\mathcal{S}}_{t_1 \dots t_n} = \{\, e \in \mathrm{Exec}_{\mathcal{S}} \mid t_1 \dots t_n \precsim e \}.$$

Note that $B^{\mathcal{S}}_{\epsilon} = \mathrm{Exec}_{\mathcal{S}}$. In the probabilistic transition system of Example 3.1.2, the basic cylinder set of t_{01} is $B^{\mathcal{S}}_{t_{01}} = \{t_{01}t_{12}, t_{01}t_{10}t_{02}, t_{01}t_{10}t_{01}t_{12}, \dots\}$.

Definition 3.3.2. The *cylinder set* $\mathcal{B}_{\mathcal{S}}$ is defined by

$$\mathcal{B}_{\mathcal{S}} = \{\, B^{\mathcal{S}}_{t_1 \dots t_n} \mid t_1 \dots t_n \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \,\} \cup \{\emptyset\}.$$

In the probabilistic transition system of Example 3.1.2, the cylinder set $\mathcal{B}_{\mathcal{S}} = \{\emptyset, B^{\mathcal{S}}_{\epsilon}, B^{\mathcal{S}}_{t_{01}}, B^{\mathcal{S}}_{t_{02}}, B^{\mathcal{S}}_{t_{01}t_{10}}, B^{\mathcal{S}}_{t_{01}t_{12}}, \dots\}$. Next, we show that the cylinder set $\mathcal{B}_{\mathcal{S}}$ is a semi-ring.

Proposition 3.3.3. *The collection $\mathcal{B}_{\mathcal{S}}$ is a semi-ring.*

Proof. Recall from Definition 2.1.5 that we have to prove

(i) $\emptyset \in \mathcal{B}_{\mathcal{S}}$,

(ii) if B, $B' \in \mathcal{B}_{\mathcal{S}}$ then $B \cap B' \in \mathcal{B}_{\mathcal{S}}$, and

(iii) if B, $B' \in \mathcal{B}_{\mathcal{S}}$ then there exist $B_i \in \mathcal{B}_{\mathcal{S}}$ for all $i \in \mathbb{N}$, which are pairwise disjoint such that $B \setminus B' = \bigcup_{i \in \mathbb{N}} B_i$.

By definition, we have $\emptyset \in \mathcal{B}_{\mathcal{S}}$. Hence, (i) holds. To prove (ii), we observe that

$$B^{\mathcal{S}}_{t_1\dots t_n} \cap B^{\mathcal{S}}_{t'_1\dots t'_m} = \begin{cases} \emptyset & \text{if } t_1\dots t_n \not\sim t'_1\dots t'_m \\ B^{\mathcal{S}}_{t_1\dots t_n} & \text{if } t'_1\dots t'_m \precsim t_1\dots t_n \\ B^{\mathcal{S}}_{t'_1\dots t'_m} & \text{if } t_1\dots t_n \precsim t'_1\dots t'_m \end{cases}$$

To prove (iii), we have

$B^{\mathcal{S}}_{t_1\dots t_n} \setminus B^{\mathcal{S}}_{t'_1\dots t'_m} =$

$$\begin{cases} \emptyset & \\ & \text{if } t'_1\dots t'_m \precsim t_1\dots t_n \\ B^{\mathcal{S}}_{t_1\dots t_n} & \\ & \text{if } t_1\dots t_n \not\sim t'_1\dots t'_m \\ \bigcup_{0\le k<m-n}\{B^{\mathcal{S}}_{t_1\dots t_n t'_{n+1}\dots t'_{n+k}t} \mid t \neq t'_{n+k+1} \wedge \text{source}_{\mathcal{S}}(t) = \text{target}_{\mathcal{S}}(t'_{n+k})\} & \\ & \text{if } t_1\dots t_n \precsim t'_1\dots t'_m \end{cases}$$

It is trivial that the sets $B^{\mathcal{S}}_{t_1\dots t_n t'_{n+1}\dots t'_{n+k}t}$ are pairwise disjoint. From Proposition 3.1.4 we know that the set $\{t \in T_{\mathcal{S}} \mid t \neq t'_{n+k+1} \wedge \text{source}_{\mathcal{S}}(t) = \text{target}_{\mathcal{S}}(t_n)\}$ is countable, and hence the set $\{B^{\mathcal{S}}_{t_1\dots t_n t'_{n+1}\dots t'_{n+k}t} \mid t \neq t'_{n+k+1} \wedge \text{source}_{\mathcal{S}}(t) = \text{target}_{\mathcal{S}}(t_n)\}$ is countable as well.

□

Next, we define a measure on the semi-ring $\mathcal{B}_{\mathcal{S}}$.

Definition 3.3.4. The function $\nu_{\mathcal{S}} : \mathcal{B}_{\mathcal{S}} \to [0,1]$ is defined by

$$\begin{aligned} \nu_{\mathcal{S}}(B^{\mathcal{S}}_{t_1 \ldots t_n}) &= \prod_{1 \leq i \leq n} \mathrm{prob}_{\mathcal{S}}(t_i) \\ \nu_{\mathcal{S}}(\emptyset) &= 0 \end{aligned}$$

Note that $\nu_{\mathcal{S}}(B^{\mathcal{S}}_{\epsilon}) = 1$.

Proposition 3.3.5. *The function $\nu_{\mathcal{S}}$ is a measure on $\mathcal{B}_{\mathcal{S}}$.*

Proof. We refer the reader to, for example, [29, Lemma 5.3.6]. □

According to Theorem 2.1.9, a measure on a semi-ring can be extended to a measure on a σ-algebra containing the semi-ring. We denote the extended measure by $\mu_{\mathcal{S}}$ and the σ-algebra containing the cylinder set $\mathcal{B}_{\mathcal{S}}$ by $\Sigma_{\mathcal{S}}$.

We shall exploit the measurable space $\langle \mathrm{Exec}_{\mathcal{S}}, \Sigma_{\mathcal{S}}, \mu_{\mathcal{S}} \rangle$ to capture our progress measure. In future chapters we shall use the following properties of this measurable space:

- $\Sigma_{\mathcal{S}}$ is a σ-algebra,
- $\mathcal{B}_{\mathcal{S}} \subseteq \Sigma_{\mathcal{S}}$, and
- $\nu(B) = \mu(B)$ for all basic cylinder sets $B \in \mathcal{B}_{\mathcal{S}}$.

3.4 A Search of a Probabilistic Transition System

As we already discussed in Section 3.1, we represent the randomized code to be verified by an explicit-state probabilistic model checker as a probabilistic transition

system. We represent the model checking of the randomized code as a sequence of transitions of the probabilistic transition system.

Definition 3.4.1. A *search* of a probabilistic transition system $\mathcal{S}$ is a sequence of distinct transitions $t_1, \ldots, t_n$ for some $n \geq 0$ such that $t_i \in T_{\mathcal{S}}$ for all $1 \leq i \leq n$.

In the probabilistic transition system of Example 3.1.2, the sequence of transitions t_{01}, t_{02} is such a search. Given a probabilistic transition system $\mathcal{S}$ and a search $t_1, \ldots, t_n$ of $\mathcal{S}$, those execution paths of $\mathcal{S}$ that only consist of the transitions $t_1, \ldots, t_n$ constitute the set

$$\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega) .$$

Next we shall show that this set is an element of $\Sigma_{\mathcal{S}}$ and, hence, the probabilistic measure $\mu_{\mathcal{S}}$ assigns a measure to the set. We shall relate this measure with our progress measure in Chapter 4.

Proposition 3.4.2. $(\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) \cup (\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega) \in \Sigma_{\mathcal{S}}$.

Proof. Obviously, the set $\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*$ is countable. Assume $e \in \mathrm{Exec}^*_{\mathcal{S}}$. Then $B^{\mathcal{S}}_e = \{e\}$. Since $\mathcal{B}_{\mathcal{S}} \subseteq \Sigma_{\mathcal{S}}$, we have that $\{e\} \in \Sigma_{\mathcal{S}}$. Since $\Sigma_{\mathcal{S}}$ is closed under countable unions, $\bigcup_{e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \{e\} = \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^* \in \Sigma_{\mathcal{S}}$.

Since $\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega = \bigcap_{k \in \mathbb{N}} \bigcup_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots, t_n\}^k} B^{\mathcal{S}}_e$ and $B^{\mathcal{S}}_e \in \Sigma_{\mathcal{S}}$ and $\Sigma_{\mathcal{S}}$ is closed under countable unions and intersections, we can conclude that $\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega \in \Sigma_{\mathcal{S}}$. ∎

4 A Progress Measure

We discussed in our introduction that having a measure of progress during model checking a randomized algorithm is useful. In the previous chapter, we modeled the probabilistic model checker as a probabilistic transition system and a search. In this chapter we shall introduce the definition of a progress measure.

4.1 Extension of the Search

If the model checker has searched a part of the system, it of course is not aware of the remainder of the system. To capture this, we formalize how a search can be extended.

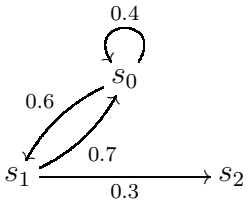
Definition 4.1.1. The probabilistic transition system $\mathcal{S}'$ *extends* the search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$ if for all $1 \leq i \leq n$,

- $t_i \in T_{\mathcal{S}'}$,
- $s_{0_{\mathcal{S}}} = s_{0_{\mathcal{S}'}}$,

- $\text{source}_{\mathcal{S}'}(t_i) = \text{source}_{\mathcal{S}}(t_i)$,
- $\text{target}_{\mathcal{S}'}(t_i) = \text{target}_{\mathcal{S}}(t_i)$,
- $\text{prob}_{\mathcal{S}'}(t_i) = \text{prob}_{\mathcal{S}}(t_i)$,
- $\text{label}_{\mathcal{S}'}(\text{source}_{\mathcal{S}'}(t_i)) = \text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t_i))$,
- $\text{label}_{\mathcal{S}'}(\text{target}_{\mathcal{S}'}(t_i)) = \text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t_i))$,
- $\text{target}_{\mathcal{S}'}(t_i)$ is final in $\mathcal{S}'$ iff $\text{target}_{\mathcal{S}}(t_i)$ is final in $\mathcal{S}$, and
- $s_{0_{\mathcal{S}}}$ is final in $\mathcal{S}'$ iff $s_{0_{\mathcal{S}'}}$ is final in $\mathcal{S}$.

Instead of $s_{0_{\mathcal{S}}}$, we shall write s_0 in the remainder of this thesis.

Example 4.1.2. Consider the probabilistic transition system of Example 3.1.2 and the search t_{01}, t_{10}. The following probabilistic transition system extends the search.



Besides the transitions t_{01} and t_{10}, the extended system has the transition t_{12} as in Example 3.1.2 and the extra transition t_{00}, but does not contain the transition t_{02}.

Note that the probabilistic transition system $\mathcal{S}$ itself extends the search $t_1, \ldots, t_n$ of $\mathcal{S}$.

Consider the search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$. Assume that the probabilistic transition system $\mathcal{S}'$ extends $t_1, \ldots, t_n$. First, we show that $\mathcal{S}$ and $\mathcal{S}'$ give rise to the same set of execution paths when we restrict to those that only consist of the transitions $t_1, \ldots, t_n$.

Proposition 4.1.3. *Let $\mathcal{S}'$ extend $t_1, \ldots, t_n$ of $\mathcal{S}$. Then*

$$\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega) = \mathrm{Exec}_{\mathcal{S}'} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega).$$

Proof. The equality can be proved by proving two inclusions. Since both parts of the proof are similar, we only show one inclusion. We distinguish the following three cases.

- If $\epsilon \in \mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)$, then $s_0 \in S_{\mathcal{S}}$ and s_0 is final in $\mathcal{S}$. Since $\mathcal{S}'$ extends $t_1, \ldots, t_n$ of $\mathcal{S}$, we know that $s_0 \in S_{\mathcal{S}'}$ and s_0 is final in $\mathcal{S}'$ as well. Hence, $\epsilon \in \mathrm{Exec}_{\mathcal{S}'}$.

- Assume that $u_1 \ldots u_m \in \mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*$. Since $u_1 \ldots u_m \in \{t_1, \ldots, t_n\}^*$ and $\mathcal{S}'$ extends $t_1, \ldots, t_n$ of $\mathcal{S}$, we have that for all $1 \leq i \leq m$,

  (1) $\mathrm{source}_{\mathcal{S}'}(u_i) = \mathrm{source}_{\mathcal{S}}(u_i)$

  (2) $\mathrm{target}_{\mathcal{S}'}(u_i) = \mathrm{target}_{\mathcal{S}}(u_i)$, and

  (3) $\mathrm{target}_{\mathcal{S}'}(u_i)$ is final in $\mathcal{S}'$ iff $\mathrm{target}_{\mathcal{S}}(u_i)$ is final in $\mathcal{S}$.

  Since $u_1 \ldots u_m \in \mathrm{Exec}_{\mathcal{S}}$, we have that

(4) $\text{source}_{\mathcal{S}}(u_{i+1}) = \text{target}_{\mathcal{S}}(u_i)$ for all $1 \leq i < m$,

(5) $\text{source}_{\mathcal{S}}(u_1) = s_0$, and

(6) $\text{target}_{\mathcal{S}}(u_m)$ is final in $\mathcal{S}$.

For all $1 \leq i < m$,

$$\begin{aligned} \text{source}_{\mathcal{S}'}(u_{i+1}) &= \text{source}_{\mathcal{S}}(u_{i+1}) \quad [(1)] \\ &= \text{target}_{\mathcal{S}}(u_i) \quad [(4)] \\ &= \text{target}_{\mathcal{S}'}(u_i) \quad [(2)]. \end{aligned}$$

Furthermore,

$$\begin{aligned} \text{source}_{\mathcal{S}'}(u_1) &= \text{source}_{\mathcal{S}}(u_1) \quad [(1)] \\ &= s_0 \quad [(5)]. \end{aligned}$$

Also,

$$\begin{aligned} & \text{target}_{\mathcal{S}}(u_m) \quad \text{is final in } \mathcal{S} \quad [(6)] \\ \text{implies that} \quad & \text{target}_{\mathcal{S}'}(u_m) \quad \text{is final in } \mathcal{S}' \quad [(2) \text{ and } (3)]. \end{aligned}$$

Hence, $u_1 \ldots u_m \in \text{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^*$.

- Assume that $u_1 u_2 \ldots \in \text{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega$. Since $u_1 u_2 \ldots \in \{t_1, \ldots, t_n\}^\omega$ and $\mathcal{S}'$ extends $t_1, \ldots, t_n$ of $\mathcal{S}$, we have that for all $i \geq 1$,

(1) $\text{source}_{\mathcal{S}'}(u_i) = \text{source}_{\mathcal{S}}(u_i)$ and

(2) $\text{target}_{\mathcal{S}'}(u_i) = \text{target}_{\mathcal{S}}(u_i)$.

Since $u_1 u_2 \ldots \in \text{Exec}_{\mathcal{S}}$, we have that

(3) $\text{source}_{\mathcal{S}}(u_{i+1}) = \text{target}_{\mathcal{S}}(u_i)$ for all $i \geq 1$ and

(4) $\text{source}_{\mathcal{S}}(u_1) = s_0$.

For all $i \geq 1$,

$$\begin{aligned} \text{source}_{\mathcal{S}'}(u_{i+1}) &= \text{source}_{\mathcal{S}}(u_{i+1}) \quad [(1)] \\ &= \text{target}_{\mathcal{S}}(u_i) \quad [(3)] \\ &= \text{target}_{\mathcal{S}'}(u_i) \quad [(2)]. \end{aligned}$$

Furthermore,

$$\begin{aligned} \text{source}_{\mathcal{S}'}(u_1) &= \text{source}_{\mathcal{S}}(u_1) \quad [(1)] \\ &= s_0 \quad [(4)]. \end{aligned}$$

Hence, $u_1 u_2 \ldots \in \text{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega$.

□

As we already showed in Proposition 3.4.2, the sets $\text{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)$ and $\text{Exec}_{\mathcal{S}'} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)$ are measurable. Next, we

show that the measures of those sets of executions are the same. In the proof, we consider the finite and infinite executions separately. But first, we prove the following simple property that will be used in both cases.

Proposition 4.1.4. *Let $\mathcal{S}'$ extend a search $t_1, \ldots, t_n$ of $\mathcal{S}$. For all $e \in \{t_1, \ldots, t_n\}^*$, we have $\nu_{\mathcal{S}}(B_e^{\mathcal{S}}) = \nu_{\mathcal{S}'}(B_e^{\mathcal{S}'})$.*

Proof. We distinguish two cases:

- if $e = \epsilon$ then

$$\nu_{\mathcal{S}}(B_\epsilon^{\mathcal{S}}) = \nu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}}) = 1 = \nu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'}) = \nu_{\mathcal{S}'}(B_\epsilon^{\mathcal{S}'}),$$

  and

- if $e = u_1 \ldots u_m$ where $u_1, \ldots, u_m \in \{t_1, \ldots, t_n\}$, then

$$\begin{aligned} & \nu_{\mathcal{S}}(B_e^{\mathcal{S}}) \\ &= \prod_{1 \le i \le m} \mathrm{prob}_{\mathcal{S}}(u_i) \\ &= \prod_{1 \le i \le m} \mathrm{prob}_{\mathcal{S}'}(u_i) \quad [\mathcal{S}' \text{ extends } t_1, \ldots, t_n \text{ of } \mathcal{S}] \\ &= \nu_{\mathcal{S}'}(B_e^{\mathcal{S}'}). \end{aligned}$$

□

Next, we show that the finite execution paths that only consist of the transitions $t_1, \ldots, t_n$ of $\mathcal{S}$ and $\mathcal{S}'$ have the same measure.

Proposition 4.1.5. *Let $\mathcal{S}'$ extend a search $t_1, \ldots, t_n$ of $\mathcal{S}$. Then*

$$\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) = \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^*).$$

Proof.

$$\begin{aligned}
&\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) \\
&= \mu_{\mathcal{S}}\left(\bigcup_{e \in \mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \{e\}\right) \\
&= \mu_{\mathcal{S}}\left(\bigcup_{e \in \{t_1, \ldots, t_n\}^*} B_e^{\mathcal{S}}\right) \\
&= \sum_{e \in \{t_1, \ldots, t_n\}^*} \mu_{\mathcal{S}}(B_e^{\mathcal{S}}) \quad [\mu_{\mathcal{S}} \text{ is a measure}] \\
&= \sum_{e \in \{t_1, \ldots, t_n\}^*} \nu_{\mathcal{S}}(B_e^{\mathcal{S}}) \quad [\mu_{\mathcal{S}} \text{ extends } \nu_{\mathcal{S}}] \\
&= \sum_{e \in \{t_1, \ldots, t_n\}^*} \nu_{\mathcal{S}'}(B_e^{\mathcal{S}'}) \quad [\mathcal{S}' \text{ extends } t_1, \ldots, t_n \text{ of } \mathcal{S} \text{ and Proposition 4.1.4}] \\
&= \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^*) \quad [\text{by symmetric argument}].
\end{aligned}$$

□

The same holds for the infinite execution paths that only consist of the transitions $t_1, \ldots, t_n$ as well.

Proposition 4.1.6. *Let $\mathcal{S}'$ extend a search $t_1, \ldots, t_n$ of $\mathcal{S}$. Then*

$$\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega) = \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega).$$

Proof. Obviously, for all $e \in \{t_1, \ldots, t_n\}^*$ and $t \in \{t_1, \ldots, t_n\}$, we have $B^{\mathcal{S}}_e \supseteq B^{\mathcal{S}}_{et}$. Hence $B^{\mathcal{S}}_\epsilon \supseteq (B^{\mathcal{S}}_{t_1} \cup \ldots \cup B^{\mathcal{S}}_{t_n}) \supseteq (B^{\mathcal{S}}_{t_1 t_1} \cup B^{\mathcal{S}}_{t_1 t_2} \cup \ldots \cup B^{\mathcal{S}}_{t_n t_n}) \supseteq \ldots$ and $\mu_{\mathcal{S}}(B^{\mathcal{S}}_\epsilon) = 1$ is finite. Then,

$$
\begin{aligned}
&\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega) \\
&= \mu_{\mathcal{S}} \left(\bigcap_{k \in \mathbb{N}} \bigcup_{u_1 \ldots u_k \in \{t_1, \ldots, t_n\}^k} B^{\mathcal{S}}_{u_1 \ldots u_k} \right) \\
&= \lim_{k \in \mathbb{N}} \mu_{\mathcal{S}} \left(\bigcup_{u_1 \ldots u_k \in \{t_1, \ldots, t_n\}^k} B^{\mathcal{S}}_{u_1 \ldots u_k} \right) \quad [\text{Proposition 2.1.8}] \\
&= \lim_{k \in \mathbb{N}} \sum_{u_1 \ldots u_k \in \{t_1, \ldots, t_n\}^k} \mu_{\mathcal{S}}(B^{\mathcal{S}}_{u_1 \ldots u_k}) \quad [\mu_{\mathcal{S}} \text{ is a measure}] \\
&= \lim_{k \in \mathbb{N}} \sum_{u_1 \ldots u_k \in \{t_1, \ldots, t_n\}^k} \nu_{\mathcal{S}}(B^{\mathcal{S}}_{u_1 \ldots u_k}) \quad [\mu_{\mathcal{S}} \text{ extends } \nu_{\mathcal{S}}] \\
&= \lim_{k \in \mathbb{N}} \sum_{u_1 \ldots u_k \in \{t_1, \ldots, t_n\}^k} \nu_{\mathcal{S}'}(B^{\mathcal{S}'}_{u_1 \ldots u_k}) \quad [\mathcal{S}' \text{ extends } t_1, \ldots, t_n \text{ of } \mathcal{S} \text{ and Proposition 4.1.4}] \\
&= \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega) \quad [\text{by symmetric argument}].
\end{aligned}
$$

□

4.2 Definition of Progress

In our definition of a probabilistic transition system, we have a set of atomic propositions. Furthermore, the definition of probabilistic transition system also contains a label function which tells us for each state which atomic propositions hold in that state. This allows us to specify properties of the system. In this thesis, we particularly focus on linear time properties. These are introduced next. In the definition

below, we use 2^{AP} to denote the set of all subsets of AP.

Definition 4.2.1. A linear time property ϕ is a set of finite or infinite sequences over the alphabet 2^{AP}.

Next, we give the definition of how an execution path satisfies a linear time property.

Definition 4.2.2. Let ϕ be a linear time property and $e \in \text{Exec}_{\mathcal{S}}$. We define $e \models_{\mathcal{S}} \phi$ as follows.

- $t_1 \dots t_n \models_{\mathcal{S}} \phi$ if $\text{label}_{\mathcal{S}}(s_0)\text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t_1)) \dots \text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t_n)) \in \phi$.
- $t_1 t_2 \dots \models_{\mathcal{S}} \phi$ if $\text{label}_{\mathcal{S}}(s_0)\text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t_1))\text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t_2)) \dots \in \phi$.

Linear temporal logic (LTL) [1, Chapter 5] is a logic to express some linear time properties. Formulas of this logic can be built from atomic propositions and operators such as $\bigcirc$, $\Box$ and $\Diamond$. The atomic proposition p corresponds to the linear time property consisting of all those sequences of subsets of AP the first element of which contains p. The formula $\bigcirc p$ corresponds to the linear time property consisting of all those sequences of subsets of AP the second element of which contains p. Similarly, the formula $\bigcirc\bigcirc p$ corresponds to the linear time property consisting of all those sequences of subsets of AP the third element of which contains p. The formula $\Box p$ corresponds to the linear time property consisting of all those

sequences of subsets of AP, all elements of which contain p. The formula $\Diamond p$ corresponds to the linear time property consisting of all those sequences of subsets of AP, one of the elements of which contains p. For more details, we refer the reader to [1, Section 5.1].

In Definition 4.2.8, we shall define the amount of progress a search has made towards verifying a linear time property. This amount only makes sense as long as no violation of the property has been found. The latter is formalized next.

Definition 4.2.3. The search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$ *has not found a violation* of the linear time property ϕ if

1. for all $e \in \{t_1, \ldots, t_n\}^\omega$, if $e \in \mathrm{Exec}_{\mathcal{S}}^\omega$ then $e \models_{\mathcal{S}} \phi$, and

2. for all $e \in \{t_1, \ldots, t_n\}^*$, if $e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}})$ then there exists a probabilistic transition system $\mathcal{S}'$ that extends $t_1, \ldots, t_n$ such that $e' \models_{\mathcal{S}'} \phi$ for all $e' \in B_e^{\mathcal{S}'}$.

Note that the search $t_1, \ldots, t_n$ has found a violation of the linear time property ϕ if

- either we can form an infinite execution path from the transitions $t_1, \ldots, t_n$ that violates ϕ,

- or we can form a prefix e of an execution path from the transitions $t_1, \ldots, t_n$ which may give rise to a violation of ϕ no matter how we extend e.

For the probabilistic transition system of Example 3.1.2, consider the linear time property $\phi = \Box p$. Assume that p is satisfied in state s_0 and s_1, but not in s_2. Then the search t_{01}, t_{10} has not found a violation of the property $\Box p$ and the search t_{01}, t_{12} has found a violation of $\Box p$.

Consider the search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$. Assume that the probabilistic transition system $\mathcal{S}'$ extends $t_1, \ldots, t_n$. The finite execution paths that only consist of the transitions $t_1, \ldots, t_n$ of $\mathcal{S}$ and $\mathcal{S}'$ satisfy the same linear time properties.

Proposition 4.2.4. *Let $\mathcal{S}'$ extend a search $t_1, \ldots, t_n$ of $\mathcal{S}$. For all $e \in \mathrm{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*$, $e \models_{\mathcal{S}} \phi$ iff $e \models_{\mathcal{S}'} \phi$.*

Proof. This can be proved along the lines of, for example, the proof of [1, Theorem 3.15].

□

Note that in our proofs, we only need the following weaker condition.

Proposition 4.2.5. *If the search $t_1, \ldots, t_n$ of $\mathcal{S}$ has not found a violation of ϕ, then $e \models_{\mathcal{S}} \phi$ for all $e \in \mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)$.*

Proof. Let $e \in \mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)$. We distinguish the following two cases.

- Assume that $e \in \text{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*$. From 2. of Definition 4.2.3 we can conclude that there exists a probabilistic transition system $\mathcal{S}'$ that extends $t_1, \ldots, t_n$ such that $e' \models_{\mathcal{S}'} \phi$ for all $e' \in B_e^{\mathcal{S}'}$. From Proposition 4.1.3 we can deduce that $e \in \text{Exec}_{\mathcal{S}'}$. Hence, $e \in B_e^{\mathcal{S}'}$ and, therefore, $e \models_{\mathcal{S}'} \phi$. From Proposition 4.2.4 we can conclude that $e \models_{\mathcal{S}} \phi$.

- When $e \in \text{Exec}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega$, then we can conclude from 1. of Definition 4.2.3 that $e \models_{\mathcal{S}} \phi$.

□

In the definition of our progress measure, we make use of the following sets.

Definition 4.2.6. Let the probabilistic transition system $\mathcal{S}'$ extend the search $t_1, \ldots, t_n$ of probabilistic transition system $\mathcal{S}$ and let ϕ be a linear time property. The set $\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n)$ is defined by

$$\begin{aligned} &\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n) \\ &\quad = \bigcup \{\, B_e^{\mathcal{S}'} \in \mathcal{B}_{\mathcal{S}'} \mid e \in \{t_1, \ldots, t_n\}^* \wedge \forall e' \in B_e^{\mathcal{S}} : e' \models_{\mathcal{S}'} \phi \,\}. \end{aligned}$$

The set $\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n)$ contains a basic cylinder set $B_e^{\mathcal{S}}$ if all its execution paths, that is, all extensions of e, satisfy ϕ. In that case, $B_e^{\mathcal{S}}$ does not contain a counterexample of ϕ. Hence, the set $\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n)$ consists of those basic cylinder sets that do not contain a counterexample of ϕ. This set is measurable, as is stated in the following proposition.

Proposition 4.2.7. *Let ϕ be a linear time property, and $\mathcal{S}'$ extend a search $t_1, \ldots, t_n$ of $\mathcal{S}$. Then $\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n) \in \Sigma_{\mathcal{S}'}$.*

Proof. For each $e \in \{t_1, \ldots, t_n\}^*$, we have that $B^{\mathcal{S}'}_e \in \mathcal{B}_{\mathcal{S}'}$, and, hence $B^{\mathcal{S}'}_e \in \Sigma_{\mathcal{S}'}$. The set $\{\, B^{\mathcal{S}'}_e \in \mathcal{B}_{\mathcal{S}'} \mid e \in \{t_1, \ldots, t_n\}^* \wedge \forall e' \in B^{\mathcal{S}'}_e : e' \models_{\mathcal{S}'} \phi \,\}$ is countable since the set $\{t_1, \ldots, t_n\}^*$ is countable. Since $\Sigma_{\mathcal{S}'}$ is a σ-algebra and, hence, closed under countable unions, the desired result follows. □

Since the set $\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n)$ is measurable, its measure $\mu_{\mathcal{S}'}(\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n))$ is defined. The latter is a number in the interval $[0, 1]$ which represents the "size" of the basic cylinder sets that do not contain a counterexample of ϕ. This number captures the amount of progress of $t_1, \ldots, t_n$ for ϕ, *provided that* the probabilistic transition system under consideration is $\mathcal{S}'$. Since we have no knowledge of the transitions other than the search, we consider that all extensions $\mathcal{S}'$ of $t_1, \ldots, t_n$ and consider the worst case in terms of progress.

Definition 4.2.8. Consider the search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$. Let ϕ be a linear time property. Assume that $t_1, \ldots, t_n$ has not found a violation of ϕ. The *progress* of the search $t_1, \ldots, t_n$ of ϕ is defined by

$$\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \phi) = \inf \left\{ \mu_{\mathcal{S}'} \left(\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n) \right) \mid \mathcal{S}' \text{ extends } t_1, \ldots, t_n \text{ of } \mathcal{S} \right\}.$$

Consider the probabilistic transition system of Example 3.1.2. Assume that $p \in \mathrm{label}_{\mathcal{S}}(s_i)$ for all $0 \leq i \leq 2$. The following table provides some examples of our

progress measure for several linear time properties and searches.

$t_1, \ldots, t_n$	$\bigcirc p$	$\bigcirc \bigcirc p$	$\Box p$	$\Diamond p$
ϵ	0.0	0.0	0.0	0.0
t_{01}	0.6	0.0	0.0	1.0
t_{02}	0.4	0.0	0.4	1.0
t_{01}, t_{02}	1.0	0.0	0.4	1.0
t_{01}, t_{12}	0.6	0.18	0.18	1.0
t_{01}, t_{10}	0.6	0.42	0.0	1.0
t_{01}, t_{10}, t_{12}	0.6	0.6	0.31	1.0
t_{01}, t_{10}, t_{02}	1.0	0.42	0.69	1.0
t_{01}, t_{12}, t_{02}	1.0	0.18	0.58	1.0

If, instead we consider the best case, by replacing the inf with a sup in the definition of our progress measure, then we conjecture that we always get one. Intuitively, since we have not found a violation of the property yet, there exists an extension of the search such that the property is always satisfied.

Conjecture 4.2.9. *Consider the search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$. Let ϕ be a linear time property. Assume that $t_1, \ldots, t_n$ has not found a violation of ϕ. Then*

$$\sup \left\{ \mu_{\mathcal{S}'} \left(\mathcal{B}^{\phi}_{\mathcal{S}'}(t_1, \ldots, t_n) \right) \mid \mathcal{S}' \text{ extends } t_1, \ldots, t_n \text{ of } \mathcal{S} \right\} = 1.$$

As we shall show in the next proposition, our progress measure gives a lower bound for the probability that the linear time property holds. For example, if the progress measure is 0.9, we know that the probability that the property holds is at least 0.9.

Proposition 4.2.10. *Consider the search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$. Let ϕ be a linear time property. Assume that $t_1, \ldots, t_n$ has not found a violation of ϕ. Then*

$$\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \phi) \leq \mu_{\mathcal{S}}(\{\, e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi \,\}).$$

Proof. From the definition of prog, Proposition 2.1.7, and the fact that $\mathcal{S}$ extends $t_1, \ldots, t_n$ of $\mathcal{S}$, we can conclude that it suffices to show that $\mathcal{B}^{\phi}_{\mathcal{S}}(t_1, \ldots, t_n)$ is a subset of $\{\, e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi \,\}$. Let $e \in \{t_1, \ldots, t_n\}^*$ and assume that $\forall e' \in B^{\mathcal{S}}_e : e' \models_{\mathcal{S}} \phi$. It suffices to show that $B^{\mathcal{S}}_e$ is a subset of $\{\, e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi \,\}$. Let $e' \in B^{\mathcal{S}}_e$. Then $e' \in \mathrm{Exec}_{\mathcal{S}}$ and $e' \models_{\mathcal{S}} \phi$. □

4.3 Characterization of Progress for Invariants

In this section, we restrict our attention to the case that the linear time property is an invariant, that is, it is of the form $\Box p$. This is an important class of properties. In particular, for actual code this class plays a key role. For example, we may want to check that the code never gives rise to any uncaught exceptions, or that it

never causes overflow. These types of properties are all expressed as invariants. We provide a characterization of our progress measure for invariants. This characterization is the basis for the algorithm to compute the progress measure for invariants presented later in this chapter. Given a search $t_1, \ldots, t_n$ of the probabilistic transition system $\mathcal{S}$, we show that the progress of $t_1, \ldots, t_n$ for an invariant for which no violation has been found yet is the measure of the set

$$\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega).$$

We prove this characterization by showing two inequalities.

Lemma 4.3.1. *Assume that the search $t_1, \ldots, t_n$ has not found a violation of $\Box p$. Then*

$$\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \Box p) \leq \mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)).$$

Proof. We shall construct a probabilistic transition system $\mathcal{S}'$ that extends $t_1, \ldots, t_n$ of $\mathcal{S}$ such that

$$\mu_{\mathcal{S}'}\left(\mathcal{B}^{\Box p}_{\mathcal{S}'}(t_1, \ldots, t_n)\right) \leq \mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)).$$

We construct the system $\mathcal{S}'$ by

- adding the extra state $s_\perp$ with $p \notin \mathrm{label}_{\mathcal{S}'}(s_\perp)$,

- adding the transition $t_\perp$ with $\text{source}_{\mathcal{S}'}(t_\perp) = \text{target}_{\mathcal{S}'}(t_\perp) = s_\perp$ and $\text{prob}_{\mathcal{S}'}(t_\perp) = 1$, and

- adding the extra transition t_s with $\text{source}_{\mathcal{S}'}(t_s) = s$, $\text{target}_{\mathcal{S}'}(t_s) = s_\perp$, and $\text{prob}_{\mathcal{S}'}(t_s) = 1 - \text{out}_{\mathcal{S}}(s)$ for all states $s \in \bigcup_{1 \le i \le n} \{\text{source}_{\mathcal{S}'}(t_i), \text{target}_{\mathcal{S}'}(t_i)\}$ such that $\text{out}_{\mathcal{S}}(s) < 1$ and s is not final in $S_{\mathcal{S}'}$, where

$$\text{out}_{\mathcal{S}}(s) = \sum \{\, \text{prob}_{\mathcal{S}}(t_i) \mid 1 \le i \le n \wedge \text{source}_{\mathcal{S}}(t_i) = s \,\}.$$

So the probabilistic transition system $\mathcal{S}'$ is defined by

- $S_{\mathcal{S}'} = \bigcup_{1 \le i \le n} \{\text{source}_{\mathcal{S}'}(t_i), \text{target}_{\mathcal{S}'}(t_i)\} \cup \{s_0, s_\perp\}$,

- $T_{\mathcal{S}'} = \{t_1, \ldots, t_n\} \cup \{\, t_s \mid s \in S_{\mathcal{S}'} \setminus \{s_\perp\} \wedge \text{out}_{\mathcal{S}}(s) < 1 \wedge s \text{ is not final in } \mathcal{S} \,\} \cup \{t_\perp\}$,

- $\text{source}_{\mathcal{S}'}(t) = \begin{cases} \text{source}_{\mathcal{S}}(t) & \text{if } t \in \{t_1, \ldots, t_n\} \\ s & \text{if } t = t_s \\ s_\perp & \text{if } t = t_\perp \end{cases}$

- $\text{target}_{\mathcal{S}'}(t) = \begin{cases} \text{target}_{\mathcal{S}}(t) & \text{if } t \in \{t_1, \ldots, t_n\} \\ s_\perp & \text{if } t = t_\perp \text{ or } t = t_s \end{cases}$

- $\text{prob}_{\mathcal{S}'}(t) = \begin{cases} \text{prob}_{\mathcal{S}}(t) & \text{if } t \in \{t_1, \ldots, t_n\} \\ 1 - \text{out}_{\mathcal{S}}(s) & \text{if } t = t_s \\ 1 & \text{if } t = t_\perp \end{cases}$

It is easy to verify that $\mathcal{S}'$ is a probabilistic transition system.

From Proposition 4.1.3, 4.1.5, 4.1.6,and 2.1.7, it suffices to show that $\mathcal{B}^{\Box p}_{\mathcal{S}'}(t_1, \ldots, t_n)$ is a subset of $\text{Exec}_{\mathcal{S}'} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)$.

Let $e \in \{t_1, \ldots, t_n\}^*$ such that $e' \models_{\mathcal{S}'} \Box p$ for all $e' \in B^{\mathcal{S}'}_e$. Let $e'' \in B^{\mathcal{S}'}_e$. Then $e'' \in \text{Exec}_{\mathcal{S}'}$ and $e'' \models_{\mathcal{S}'} \Box p$. From the construction of $\mathcal{S}'$ we can conclude that $t_\perp$ is not part of e''. Hence, $e'' \in \{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega$. □

The key step of proving the other inequality is proving that the set Z defined below has measure zero.

Definition 4.3.2. Let $\mathcal{S}'$ extend $t_1, \ldots, t_n$ of $\mathcal{S}$. The set Z is defined by

$$Z = (\text{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega) \setminus \mathcal{B}^{\Box p}_{\mathcal{S}'}(t_1, \ldots, t_n).$$

Let $\mathcal{S}$ be the probabilistic transition system of Example 3.1.2, and let $\mathcal{S}'$ be the probabilistic transition system of Example 4.1.2. Assume that $p \in \text{label}_{\mathcal{S}'}(s_0), \text{label}_{\mathcal{S}'}(s_1)$ and $p \notin \text{label}_{\mathcal{S}'}(s_2)$. Then $\mathcal{S}'$ extends the search t_{01}, t_{10} of $\mathcal{S}$. In that case $Z = \{(t_{01}t_{10})^\omega\}$.

In the proof that the set Z has measure zero we use the following set.

Definition 4.3.3. Let $e \in \text{pref}(Z)$ and $n, m \in \mathbb{N}$. The set $Z(e, n, m)$ is defined by

$$Z(e, n, m) = \{\, \bar{e}[n + m] \mid \bar{e} \in Z \wedge \bar{e}[n] = e[n] \,\}.$$

For the probabilistic transition system of Example 3.1.2 and the search t_{01}, t_{10}, we have that $Z = \{(t_{01}t_{10})^\omega\}$ and $Z(t_{01}, 1, 2) = \{t_{01}t_{10}\}$. Here we need to exploit some properties of the sets Z and $Z(e, n, m)$. Next we present a series of technical results.

Proposition 4.3.4. *For all* $e \in \text{pref}(Z)$, $n, m \in \mathbb{N}$, *and* e_1, $e_2 \in Z(e, n, m)$, *if* $e_1 \neq e_2$ *then* $e_1 \not\sim e_2$.

Proof. It suffices to prove that $e_1 \not\precsim e_2$.

Since $e_1 \in Z(e, n, m)$, $e_1 = \bar{e}[n + m]$ for some $\bar{e} \in Z$ such that $\bar{e}[n] = e[n]$. Towards a contradiction, assume that $e_1 \precsim e_2$. As a consequence, target(e_1) is not final. Hence, $|e_1| = n + m$. Since $e_2 \in Z(e, n, m)$ and $e_1 \precsim e_2$, $|e_2| = n + m$ and, therefore, $e_1 = e_2$, which contradicts that $e_1 \neq e_2$.

□

From the above result, we can immediately conclude the following result.

Corollary 4.3.5. *For all* $e \in \text{pref}(Z)$, n, $m \in \mathbb{N}$, *and* e_1, $e_2 \in Z(e, n, m)$, *if* $e_1 \neq e_2$ *then* $B^{\mathcal{S}}_{e_1} \cap B^{\mathcal{S}}_{e_2} = \emptyset$.

Proposition 4.3.6. *For all* $e \in \text{pref}(Z)$, $n, m \in \mathbb{N}$ *and* e_1, $e_2 \in Z(\epsilon, 0, n)$, *if* $e_1 \neq e_2$ *then*

$$\left(\bigcup_{\bar{e}_1 \in Z(e_1, n, m)} B^{\mathcal{S}}_{\bar{e}_1} \right) \cap \left(\bigcup_{\bar{e}_2 \in Z(e_2, n, m)} B^{\mathcal{S}}_{\bar{e}_2} \right) = \emptyset.$$

Proof. Let $\bar{e}_1 \in Z(e_1, n, m)$. Then

$$\bar{e}_1 = \hat{e}_1[n+m] \tag{4.1}$$

for some $\hat{e}_1 \in Z$ such that

$$\hat{e}_1[n] = e_1[n]. \tag{4.2}$$

Let $\bar{e}_2 \in Z(e_2, n, m)$. Then

$$\bar{e}_2 = \hat{e}_2[n+m] \tag{4.3}$$

for some $\hat{e}_2 \in Z$ such that

$$\hat{e}_2[n] = e_2[n]. \tag{4.4}$$

Since $e_1, e_2 \in Z(\epsilon, 0, n)$, we have $e_1[n] = e_1$ and $e_2[n] = e_2$. Because $e_1 \neq e_2$, we know $e_1 \not\sim e_2$ by Proposition 4.3.4. Thus $e_1[n] \not\sim e_2[n]$. Hence by (4.2) and (4.4) we can conclude that $\hat{e}_1[n] \not\sim \hat{e}_2[n]$. Therefore $\hat{e}_1[n+m] \not\sim \hat{e}_2[n+m]$. Then by (4.1) and (4.3), $\bar{e}_1 \not\sim \bar{e}_2$, which implies that $B^{\mathcal{S}}_{\bar{e}_1} \cap B^{\mathcal{S}}_{\bar{e}_2} = \emptyset$.

□

Proposition 4.3.7. *For all n, $m \in \mathbb{N}$,*

$$\bigcup_{e \in Z(\epsilon,0,n)} \bigcup_{e' \in Z(e,n,m)} B^{\mathcal{S}}_{e'} = \bigcup_{\bar{e} \in Z(\epsilon,0,n+m)} B^{\mathcal{S}}_{\bar{e}}.$$

Proof. We prove two inclusions.

- Let $e \in Z(\epsilon, 0, n)$. Then,

$$e = \hat{e}[n] \tag{4.5}$$

for some $\hat{e} \in Z$. Let $e' \in Z(e, n, m)$. Then,

$$e' = \tilde{e}[n+m] \tag{4.6}$$

for some $\tilde{e} \in Z$ such that

$$\tilde{e}[n] = e[n]. \tag{4.7}$$

We have to prove $B^{\mathcal{S}}_{e'} \subseteq \bigcup_{\bar{e} \in Z(\epsilon,0,n+m)} B^{\mathcal{S}}_{\bar{e}}$. It suffices to show that $e' \in Z(\epsilon, 0, n+m)$. This follows immediately from (4.6) and the fact that $\tilde{e} \in Z$.

- Let $\bar{e} \in Z(\epsilon, 0, n+m)$. Then,

$$\bar{e} = \hat{e}[n+m] \tag{4.8}$$

for some $\hat{e} \in Z$. In order to show that

$$B^{\mathcal{S}}_{\bar{e}} \subseteq \bigcup_{e \in Z(\epsilon,0,n)} \bigcup_{e' \in Z(e,n,m)} B^{\mathcal{S}}_{e'}$$

it suffices to show that $\bar{e} \in Z(e, n, m)$ for some $e \in Z(\epsilon, 0, n)$. Since $\hat{e} \in Z$, we have $\hat{e}[n] \in Z(\epsilon, 0, n)$ and from (4.8) we conclude that $\bar{e} \in Z(\hat{e}[n], n, m)$.

□

Next, we introduce the function choose. Given a state s, the function returns a set of transitions. This set can either be empty or contain a single transition. If the set contains the transition t, then $\text{source}_{\mathcal{S}}(t) = s$, t is different from $t_1, \ldots, t_n$, and there exists a prefix e of an execution path from the initial state to state s

consisting only of transitions in $t_1, \ldots, t_n$ such that et is a prefix of an execution that violates the invariant. If no such transition t exists, then the set is empty.

Definition 4.3.8. The function $\text{out}_{\neg\Box p} : S_{\mathcal{S}'} \to 2^{T_{\mathcal{S}'}}$ is defined by

$$\begin{aligned}&\text{out}_{\neg\Box p}(s)\\ =\ &\{t \in T_{\mathcal{S}'} \setminus \{t_1, \ldots, t_n\} \mid \exists e \in \{t_1, \ldots, t_n\}^* : \text{target}_{\mathcal{S}'}(e) = s \wedge \exists e' \in B^{\mathcal{S}}_{et} : e' \not\models_{\mathcal{S}'} \Box p\}.\end{aligned}$$

The function $\text{choose} : S_{\mathcal{S}'} \to 2^{T_{\mathcal{S}'}}$ is defined by

$$\text{choose}(s) = \begin{cases} \emptyset & \text{if } \text{out}_{\neg\Box p}(s) = \emptyset \\ \{t\} & \text{if } \text{out}_{\neg\Box p}(s) \neq \emptyset \end{cases}$$

where t is chosen arbitrarily from $\text{out}_{\neg\Box p}(s)$.

Furthermore, we introduce a set of transitions Δ.

Definition 4.3.9. The set Δ is defined by

$$\Delta = \bigcup_{s \in \{\text{target}_{\mathcal{S}'}(t_i) \mid 1 \leq i \leq n\} \cup \{s_0\}} \text{choose}(s) \cup \{t_1, \ldots, t_n\}.$$

Proposition 4.3.10. *If $Z \neq \emptyset$ then $\Delta \neq \emptyset$.*

Proof. We distinguish the following three cases.

- If $n \neq 0$ then the set Δ is nonempty since $t_1 \in \Delta$.
- Towards a contradiction, assume that $n = 0$ then $\{t_1, \ldots, t_n\}^\omega = \emptyset$ and, hence, $Z = \emptyset$. This contradicts our assumption that $Z \neq \emptyset$.

□

Note that the set Δ is finite, we define the value δ as the minimal probability of the transitions in Δ.

Definition 4.3.11. We define δ as

$$\delta = \min(\{\, \mathrm{prob}_{\mathcal{S}'}(t) \mid t \in \Delta\}).$$

Corollary 4.3.12. *If $Z \neq \emptyset$ then $\delta \in (0, 1]$.*

We need two more technical results.

Proposition 4.3.13. *For all n, $m \in \mathbb{N}$, if $e \in Z(\epsilon, 0, m)$ and t_1, ..., t_n has not found a violation of $\Box p$, then*

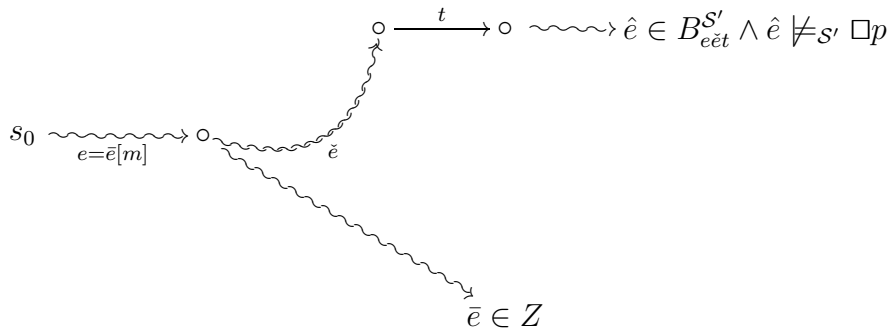
$$\mu_{\mathcal{S}'}\left(\bigcup_{e' \in Z(e,m,n+1)} B_e^{\mathcal{S}'}\right) \leq (1 - \delta^{n+1})\mu_{\mathcal{S}'}(B_e^{\mathcal{S}'}).$$

Proof. Because $e \in Z(\epsilon, 0, m)$, we have that $\bar{e}[m] = e$ for some $\bar{e} \in Z$. Thus $\bar{e} \in \{t_1, \ldots, t_n\}^\omega$, which implies that $e \in \{t_1, \ldots, t_n\}^*$.

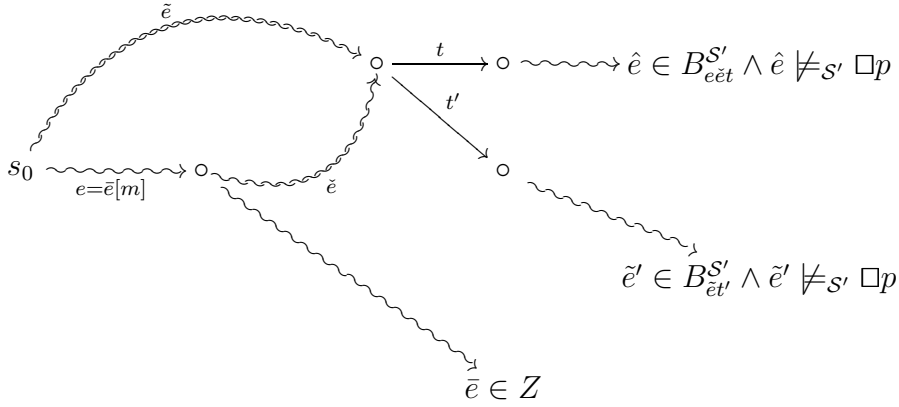
$$s_0 \underset{e=\bar{e}[m]}{\rightsquigarrow} \circ \rightsquigarrow \bar{e} \in Z$$

Since $\bar{e} \in B_e^{\mathcal{S}'}$ and $\bar{e} \in Z$, it cannot be the case that $\tilde{e} \models_{\mathcal{S}'} \Box p$ for all $\tilde{e} \in B_e^{\mathcal{S}'}$. Hence, $\hat{e} \not\models_{\mathcal{S}'} \Box p$ for some $\hat{e} \in B_e^{\mathcal{S}'}$. Since t_1, ..., t_n has not found a violation of $\Box p$, we can conclude that $\hat{e}$ contains at least one transition not in $\{t_1, \ldots, t_n\}$. Let

t be the first transition in $\hat{e}$ such that $t \notin \{t_1, \ldots, t_n\}$. Let $\check{e} \in \{t_1, \ldots, t_n\}^*$ be such that $\hat{e} \in B^{\mathcal{S}'}_{e\check{e}t}$.

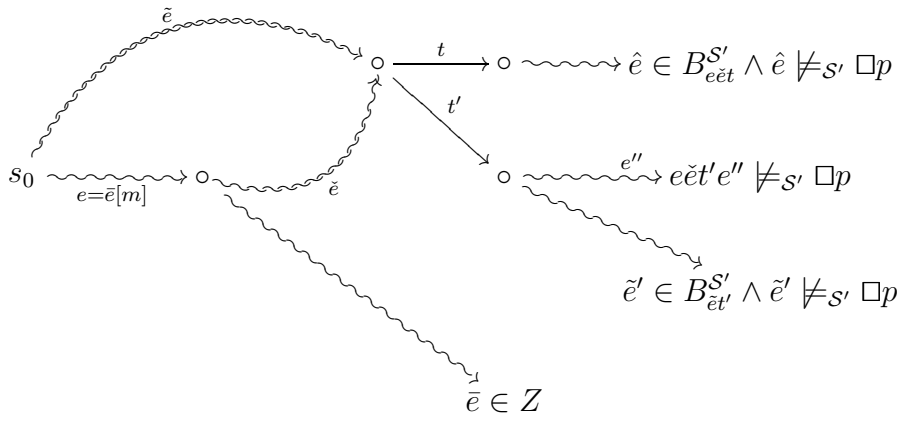


Since $\hat{e} \not\models_{\mathcal{S}'} \Box p$, we have that $t \in \text{out}_{\neg\Box p}(\text{source}_{\mathcal{S}'}(t))$. Hence, $\text{out}_{\neg\Box p}(\text{source}_{\mathcal{S}'}(t)) \neq \emptyset$. Assume that $\text{choose}(\text{source}_{\mathcal{S}'}(t)) = \{t'\}$. Since $t' \in \text{out}_{\neg\Box p}(\text{source}_{\mathcal{S}'}(t))$, there exists $\tilde{e} \in \{t_1, \ldots, t_n\}^*$ and $\tilde{e}' \in B^{\mathcal{S}'}_{\tilde{e}t'}$ such that $\tilde{e}' \not\models_{\mathcal{S}'} \Box p$.

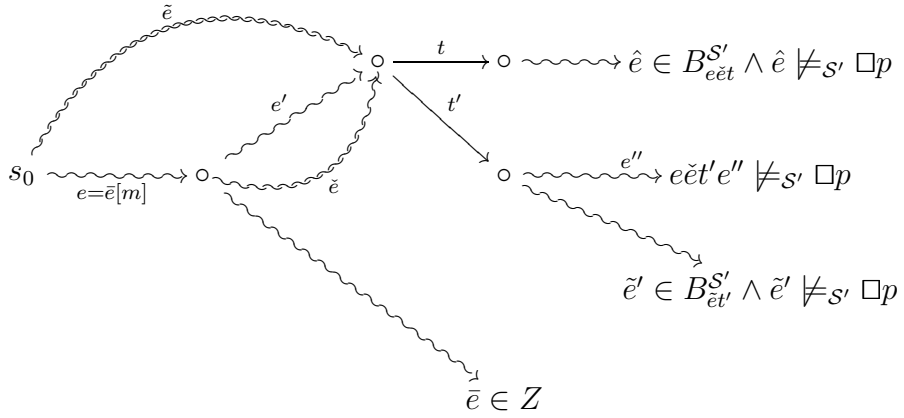


Because $t_1 \ldots t_n$ has not found a violation of $\Box p$ and $\tilde{e}$, $e\check{e} \in \{t_1, \ldots, t_n\}^*$ and $\tilde{e}' \not\models_{\mathcal{S}'} \Box p$, we can conclude that there exists a e'' such that $e\check{e}t'e'' \in B^{\mathcal{S}'}_{e\check{e}t'}$ and

$e\check{e}t'e'' \not\models_{\mathcal{S}'} \Box p$.



Since $\check{e} \in \{t_1, \ldots, t_n\}^*$, there exists a subsequence e' of $\check{e}$ from $\mathrm{target}_{\mathcal{S}'}(e)$ to $\mathrm{target}_{\mathcal{S}'}(e\check{e})$ with $|e'| \leq n$ such that $ee't'e'' \in B^{\mathcal{S}'}_e$.



Thus we have

$$\begin{aligned} \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{ee't'}) &\geq \delta^{|e't'|}\mu_{\mathcal{S}'}(B^{\mathcal{S}'}_e) \quad [e't' \in \Delta^*] \\ &\geq \delta^{n+1}\mu_{\mathcal{S}'}(B^{\mathcal{S}'}_e) \quad [|e't'| \leq n+1] \end{aligned} \tag{4.9}$$

Since $B^{\mathcal{S}'}_{ee't'} \subseteq B^{\mathcal{S}'}_e$ and $B^{\mathcal{S}'}_{\dot{e}} \subseteq B^{\mathcal{S}'}_e$ for all $\dot{e} \in Z(e, m, n+1)$ we have

$$\left(\bigcup_{\dot{e} \in Z(e,m,n+1)} B^{\mathcal{S}'}_{\dot{e}} \right) \cup B^{\mathcal{S}'}_{ee't'} \subseteq B^{\mathcal{S}}_e \tag{4.10}$$

Furthermore, for all $\dot{e} \in Z(e, m, n+1)$,

$$B^{\mathcal{S}'}_{\dot{e}} \cap B^{\mathcal{S}'}_{ee't'} = \emptyset. \tag{4.11}$$

because

- $ee't'$ cannot be a prefix of $\dot{e}$ since $t' \notin \{t_1, \ldots, t_n\}$ and $\dot{e} \in \{t_1, \ldots, t_n\}^*$ and
- $\dot{e}$ cannot be a strict prefix of $ee't'$ because $|\dot{e}| = m + n + 1 \geq |ee't'|$.

Thus we have

$$\begin{aligned}
&\mu_{\mathcal{S}'}\left(\bigcup_{\dot{e} \in Z(e,m,n+1)} B^{\mathcal{S}'}_{\dot{e}}\right) + \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{ee't'}) \\
&= \mu_{\mathcal{S}'}\left(\bigcup_{\dot{e} \in Z(e,m,n+1)} B^{\mathcal{S}'}_{\dot{e}} \cup B^{\mathcal{S}'}_{ee't'}\right) \quad [(4.11)] \\
&\leq \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e}) \quad [\text{Proposition 2.1.7 and } (4.10)]
\end{aligned}$$

and, hence,

$$\begin{aligned}
\mu_{\mathcal{S}'}\left(\bigcup_{\dot{e} \in Z(e,m,n+1)} B^{\mathcal{S}'}_{\dot{e}}\right) &\leq \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e}) - \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{ee't'}) \\
&\leq \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e}) - \delta^{n+1}\mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e}) \quad [(4.9)] \\
&= (1 - \delta^{n+1})\mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e})
\end{aligned}$$

□

Proposition 4.3.14. *Assume that asearch $t_1, \ldots, t_n$ has not found a violation of $\Box p$. For all $k \in \mathbb{N}$,*

$$\mu_{\mathcal{S}'}\left(\bigcup_{e \in Z} B^{\mathcal{S}'}_{e[k(n+1)]}\right) \leq (1 - \delta^{n+1})^k.$$

Proof. We prove this property by induction on k.

- If $k = 0$ then

$$\mu_{\mathcal{S}'}\left(\bigcup_{e\in Z} B^{\mathcal{S}'}_{e[0]}\right) = \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{\epsilon}) = 1 = (1-\delta^{n+1})^0.$$

- Let $k > 0$. Assume that

$$\mu_{\mathcal{S}'}\left(\bigcup_{e\in Z} B^{\mathcal{S}'}_{e[(k-1)(n+1)]}\right) \leq (1-\delta^{n+1})^{k-1}. \tag{4.12}$$

  By Proposition 4.3.13, for all $e \in Z(\epsilon, 0, (k-1)(n+1))$,

$$\mu_{\mathcal{S}'}\left(\bigcup_{\bar{e}\in Z(e,(k-1)(n+1),n+1)} B^{\mathcal{S}'}_{\bar{e}}\right) \leq (1-\delta^{n+1})\mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e}). \tag{4.13}$$

Thus,

$$
\begin{aligned}
&\mu_{\mathcal{S}'}\left(\bigcup_{e\in Z} B^{\mathcal{S}'}_{e[k(n+1)]}\right)\\
&= \mu_{\mathcal{S}'}\left(\bigcup_{e\in Z(\epsilon,0,k(n+1))} B^{\mathcal{S}'}_{e}\right)\\
&= \mu_{\mathcal{S}'}\left(\bigcup_{e\in Z(\epsilon,0,(k-1)(n+1))}\ \bigcup_{\bar{e}\in Z(e,(k-1)(n+1),n+1)} B^{\mathcal{S}'}_{\bar{e}}\right) \quad \text{[Proposition 4.3.7]}\\
&= \sum_{e\in Z(\epsilon,0,(k-1)(n+1))} \mu_{\mathcal{S}'}\left(\bigcup_{\bar{e}\in Z(e,(k-1)(n+1),n+1)} B^{\mathcal{S}'}_{\bar{e}}\right) \quad \text{[Proposition 4.3.6]}\\
&\leq \sum_{e\in Z(\epsilon,0,(k-1)(n+1))} (1-\delta^{n+1})\mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e}) \quad \text{[(4.13)]}\\
&= (1-\delta^{n+1}) \sum_{e\in Z(\epsilon,0,(k-1)(n+1))} \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_{e})\\
&= (1-\delta^{n+1})\mu_{\mathcal{S}'}\left(\bigcup_{e\in Z(\epsilon,0,(k-1)(n+1))} B^{\mathcal{S}'}_{e}\right) \quad \text{[Corollary 4.3.5]}\\
&= (1-\delta^{n+1})\mu_{\mathcal{S}'}\left(\bigcup_{e\in Z} B^{\mathcal{S}'}_{e[(k-1)(n+1)]}\right)\\
&\leq (1-\delta^{n+1})^{k} \quad \text{[induction hypothesis (4.12)]}
\end{aligned}
$$

□

Now we show that the set of Z has measure zero.

Lemma 4.3.15. *If $t_1, \ldots, t_n$ has not found a violation of $\Box p$ then $\mu_{\mathcal{S}'}(Z) = 0$.*

Proof. We distinguish the following two cases.

- If $Z = \emptyset$, then obviously $\mu_{\mathcal{S}'}(Z) = 0$.

- Otherwise, for all $e \in Z$ we have that $e \in B^{\mathcal{S}'}_{e[k(n+1)]}$ for all $k \in \mathbb{N}$. Hence,

$$Z \subseteq \bigcap_{k \in \mathbb{N}} \left(\bigcup_{e \in Z} B^{\mathcal{S}'}_{e[k(n+1)]} \right).$$

Obviously, for all $e \in Z$ we have

$$\epsilon = e[0] \precsim e[n+1] \precsim e[2(n+1)] \precsim \ldots \precsim e[k(n+1)] \precsim \ldots$$

which implies that

$$B^{\mathcal{S}'}_{\epsilon} = B^{\mathcal{S}'}_{e[0]} \supseteq B^{\mathcal{S}'}_{e[n+1]} \supseteq B^{\mathcal{S}'}_{e[2(n+1)]} \supseteq \ldots \supseteq B^{\mathcal{S}'}_{e[k(n+1)]} \supseteq \ldots.$$

Thus

$$B^{\mathcal{S}'}_{\epsilon} = \bigcup_{e \in Z} B^{\mathcal{S}'}_{e[0]} \supseteq \bigcup_{e \in Z} B^{\mathcal{S}'}_{e[n+1]} \supseteq \bigcup_{e \in Z} B^{\mathcal{S}'}_{e[2(n+1)]} \supseteq \ldots \supseteq \bigcup_{e \in Z} B^{\mathcal{S}'}_{e[k(n+1)]} \supseteq \ldots.$$

Furthermore, $\mu_{\mathcal{S}}(B^{\mathcal{S}'}_{\epsilon}) = 1$.

We use the above to apply Proposition 2.1.8 below.

$$\begin{aligned}
\mu_{\mathcal{S}'}(Z) &\leq \mu_{\mathcal{S}'}\left(\bigcap_{k \in \mathbb{N}} \left(\bigcup_{e \in Z} B^{\mathcal{S}'}_{e[k(n+1)]} \right) \right) \quad \text{[Proposition 2.1.7]} \\
&= \lim_{k \in \mathbb{N}} \mu_{\mathcal{S}'}\left(\bigcup_{e \in Z} B^{\mathcal{S}'}_{e[k(n+1)]} \right) \quad \text{[Proposition 2.1.8]} \\
&\leq \lim_{k \in \mathbb{N}} (1 - \delta^{n+1})^k \quad \text{[Proposition 4.3.14]} \\
&\leq 0 \quad \text{[Corollary 4.3.12]}
\end{aligned}$$

□

Now, we are ready to prove the other inequality.

Lemma 4.3.16. *Assume that $t_1, \ldots, t_n$ has not found a violation of $\Box p$. Then*

$$\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)) \leq \mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \Box p).$$

Proof. It suffices to show that

$$\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)) \leq \mu_{\mathcal{S}'}(\mathcal{B}_{\mathcal{S}'}^{\Box p})$$

for each $\mathcal{S}'$ that extends the search $t_1, \ldots, t_n$ of $\mathcal{S}$.

Let $\mathcal{S}'$ extend the search $t_1, \ldots, t_n$ of $\mathcal{S}$. Since $t_1, \ldots, t_n$ has not found a violation of $\Box p$,

$$\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^* \subseteq \mathcal{B}_{\mathcal{S}'}^{\Box p}(t_1, \ldots, t_n). \tag{4.14}$$

Then

$$\begin{aligned}
&\mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)) \\
&= \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)) \quad \text{[Proposition 4.1.3, 4.1.5 and 4.1.6]} \\
&= \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^*) \\
&\quad + \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega \cap \mathcal{B}_{\mathcal{S}'}^{\Box p}(t_1, \ldots, t_n)) \\
&\quad + \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega \setminus \mathcal{B}_{\mathcal{S}'}^{\Box p}(t_1, \ldots, t_n)) \\
&= \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^*) \\
&\quad + \mu_{\mathcal{S}'}(\mathrm{Exec}_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^\omega \cap \mathcal{B}_{\mathcal{S}'}^{\Box p}(t_1, \ldots, t_n)) \quad \text{[Lemma 4.3.15]} \\
&\leq \mu_{\mathcal{S}'}\left(\mathcal{B}_{\mathcal{S}'}^{\Box p}(t_1, \ldots, t_n)\right) \quad \text{[(4.14) and Proposition 2.1.7]}.
\end{aligned}$$

□

Combining the above results, we arrive at the characterization of our progress measure for invariants.

Theorem 4.3.17. *Assume that $t_1, \ldots, t_n$ has not found a violation of $\Box p$. Then*

$$\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \Box p) = \mu_{\mathcal{S}}(\mathrm{Exec}_{\mathcal{S}} \cap (\{t_1, \ldots, t_n\}^* \cup \{t_1, \ldots, t_n\}^\omega)).$$

Proof. Immediate consequence of Lemma 4.3.1 and 4.3.16. □

Note that the above theorem does not hold for all linear time properties. For example, consider the probabilistic transition system of Example 3.1.2, the search t_{01} and the property $\bigcirc p$. As we have already seen, the progress of t_{01} for $\bigcirc p$ is 0.6. However, the set $\mathrm{Exec}_{\mathcal{S}} \cap (\{t_{01}\}^* \cup \{t_{01}\}^\omega)$ is empty and, hence, its measure is zero.

4.4 Computation of Progress for Invariants

Given the characterization of our progress measure for invariants, we are now in a position to compute the progress measure for invariants. In this section, we fix a probabilistic transition system $\mathcal{S}$, which we call the complete system, a search $t_1, \ldots, t_n$ of this system and an invariant $\Box p$. Next, we construct a probabilistic transition system $\mathcal{S}'$, which we call the searched system. This searched system

is usually considerably smaller than the complete system. We shall exploit the searched system to compute the progress of $t_1, \ldots, t_n$ for $\Box p$. First we give the definition of the searched system $\mathcal{S}'$.

Definition 4.4.1. The set $S_{\mathcal{S}'}$ is defined by

$$S_{\mathcal{S}'} = \bigcup_{1 \le i \le n} \{\text{source}_{\mathcal{S}}(t_i), \text{target}_{\mathcal{S}}(t_i)\} \cup \{s_0, s_\perp\}.$$

A state s is *partial* if s is not final in $\mathcal{S}$ and $\text{out}_{\mathcal{S}}(s) < 1$.

The set $T_{\mathcal{S}'}$ is defined by

$$T_{\mathcal{S}'} = \{t_1, \ldots, t_n\} \cup \{\, t_s \mid s \in S_{\mathcal{S}'} \text{ is final in } \mathcal{S} \text{ or partial} \,\} \cup \{t_\perp\}.$$

The set $AP_{\mathcal{S}'}$ is defined by

$$AP_{\mathcal{S}'} = AP_{\mathcal{S}} \cup \{\perp\}.$$

The function $\text{source}_{\mathcal{S}'} : T_{\mathcal{S}'} \to S_{\mathcal{S}'}$ is defined by

$$\text{source}_{\mathcal{S}'}(t) = \begin{cases} s_\perp & \text{if } t = t_\perp \\ s & \text{if } t = t_s \\ \text{source}_{\mathcal{S}}(t) & \text{if } t \in \{t_1, \ldots, t_n\}. \end{cases}$$

The function $\text{target}_{\mathcal{S}'} : T_{\mathcal{S}'} \to S_{\mathcal{S}'}$ is defined by

$$\text{target}_{\mathcal{S}'}(t) = \begin{cases} s_\perp & \text{if } t = t_\perp \\ s & \text{if } t = t_s \text{ and } s \text{ is final} \\ s_\perp & \text{if } t = t_s \text{ and } s \text{ is partial} \\ \text{target}_{\mathcal{S}}(t) & \text{if } t \in \{t_1, \ldots t_n\}. \end{cases}$$

The function $\mathrm{prob}_{\mathcal{S}'} : T_{\mathcal{S}'} \to (0,1]$ is defined by

$$\mathrm{prob}_{\mathcal{S}'}(t) = \begin{cases} 1 & \text{if } t = t_\perp \\ 1 & \text{if } t = t_s \text{ and } s \text{ is final} \\ 1 - \mathrm{out}_{\mathcal{S}}(s) & \text{if } t = t_s \text{ and } s \text{ is partial} \\ \mathrm{prob}_{\mathcal{S}}(t) & \text{if } t \in \{t_1, \ldots t_n\}. \end{cases}$$

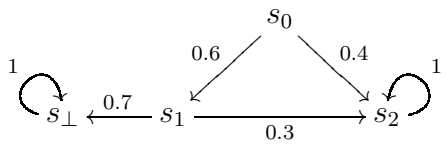
The function $\mathrm{label}_{\mathcal{S}'} : S_{\mathcal{S}'} \to 2^{AP_{\mathcal{S}'}}$ is defined by

$$\mathrm{label}_{\mathcal{S}'}(s) = \begin{cases} \{\perp\} & \text{if } s = s_\perp \\ \mathrm{label}_{\mathcal{S}}(s) & \text{otherwise.} \end{cases}$$

Obviously we have the following property.

Proposition 4.4.2. *$\mathcal{S}'$ is a probabilistic transition system.*

For the complete system of Example 3.1.2 and the search t_{01}, t_{02}, t_{12}, the corresponding searched system can be depicted as



Note that the probabilistic transition system $\mathcal{S}'$ only gives rise to infinite execution paths. Let us denote the set of execution paths of this system by $\mathrm{Exec}_{\mathcal{S}'}$. As we turned the set $\mathrm{Exec}_{\mathcal{S}}$ into a measurable space, we can also turn the set $\mathrm{Exec}_{\mathcal{S}'}$ into a measurable space $\langle \mathrm{Exec}_{\mathcal{S}'}, \Sigma_{\mathcal{S}'}, \mu_{\mathcal{S}'} \rangle$.

To prove that we can indeed use the searched system to compute the progress of the search in the complete system, we relate the complete and the searched system by linking the execution paths of the two systems.

Definition 4.4.3. The function $\eta : (\mathrm{Exec}^*_{\mathcal{S}} \cup \mathrm{Exec}^\omega_{\mathcal{S}}) \to \mathrm{Exec}_{\mathcal{S}'}$ is defined by

$$\eta(e) = \begin{cases} e(t_{\mathrm{target}_{\mathcal{S}}(e)})^\omega & \text{if } e \in \{t_1, \ldots, t_n\}^* \\ e & \text{if } e \in \{t_1, \ldots, t_n\}^\omega \\ e' t_{\mathrm{target}_{\mathcal{S}}(e')}(t_\perp)^\omega & \text{if } e = e'te'' \text{ and } e' \in \{t_1, \ldots, t_n\}^* \text{ and } t \notin \{t_1, \ldots, t_n\} \end{cases}$$

Next, we characterize those execution paths of the complete system, which solely consist of transitions of the search, in terms of execution paths in the searched system.

Proposition 4.4.4. *The function η provides a one-to-one correspondence between the sets $(\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) \cup (\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega)$ and $\{\, e' \in \mathrm{Exec}_{\mathcal{S}'} \mid e'$ does not contain $t_\perp \,\}$.*

As we show next, to compute the progress of the search in the complete system, it suffices to compute the measure of the set of those execution paths of the searched system that do not contain the transition $t_\perp$.

Theorem 4.4.5. *Assume that the search $t_1, \ldots, , t_n$ has not found a violation of $\Box p$*

$$\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \Box p) = \mu_{\mathcal{S}'}(\{\, e' \in \mathrm{Exec}_{\mathcal{S}'} \mid e' \text{ does not contain } t_\perp \}).$$

Proof. We have that

$$
\begin{aligned}
&\mu_{\mathcal{S}}(\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) \\
&= \mu_{\mathcal{S}}\left(\bigcup_{e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} B^{\mathcal{S}}_e \right) \quad [B^{\mathcal{S}}_e = \{e\} \text{ for all } e \in \mathrm{Exec}^*_{\mathcal{S}}] \\
&= \sum_{e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \mu_{\mathcal{S}}(B^{\mathcal{S}}_e) \\
&= \sum_{e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \nu_{\mathcal{S}}(B^{\mathcal{S}}_e) \quad [\mu_{\mathcal{S}} \text{ extends } \nu_{\mathcal{S}}] \\
&= \sum_{t'_1 \ldots t'_k \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \nu_{\mathcal{S}}(B^{\mathcal{S}}_{t'_1 \ldots t'_k}) \\
&= \sum_{t'_1 \ldots t'_k \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \prod_{1 \le i \le k} \mathrm{prob}_{\mathcal{S}}(t'_i) \\
&= \sum_{t'_1 \ldots t'_k \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \prod_{1 \le i \le k} \mathrm{prob}_{\mathcal{S}'}(t'_i) \quad [\mathrm{prob}_{\mathcal{S}}(t_i) = \mathrm{prob}_{\mathcal{S}'}(t_i) \text{ for all } 1 \le i \le n] \\
&= \sum_{t'_1 \ldots t'_k \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \nu_{\mathcal{S}'}(B^{\mathcal{S}'}_{t'_1 \ldots t'_k}) \\
&= \sum_{e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \nu_{\mathcal{S}'}(B^{\mathcal{S}'}_e) \\
&= \sum_{e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*} \mu_{\mathcal{S}'}(B^{\mathcal{S}'}_e) \quad [\mu_{\mathcal{S}'} \text{ extends } \nu_{\mathcal{S}'}] \\
&= \mu_{\mathcal{S}'}\left(\bigcup_{e \in \mathrm{Exec}^*_{\mathcal{S}'} \cap \{t_1, \ldots, t_n\}^*} B^{\mathcal{S}'}_e \right) \\
&= \mu_{\mathcal{S}'}(\{\, e(t_{\mathrm{target}_{\mathcal{S}'}(e)})^\omega \mid e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^* \,\}) \\
&\qquad [B^{\mathcal{S}'}_e = \{e(t_{\mathrm{target}_{\mathcal{S}'}(e)})^\omega\} \text{ for all } e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*]
\end{aligned}
$$

For all $k \in \mathbb{N}$,

$$
\bigcup_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots, t_n\}^k} B^{\mathcal{S}}_e \supseteq \bigcup_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots, t_n\}^{k+1}} B^{\mathcal{S}}_e
$$

Furthermore, $\mu_{\mathcal{S}}(B^{\mathcal{S}}_{\epsilon}) = 1$.

We use the above to apply Proposition 2.1.8 below.

$$
\begin{aligned}
&\mu_{\mathcal{S}}(\mathrm{Exec}^{\omega}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^{\omega}) \\
&= \mu_{\mathcal{S}}\left(\bigcap_{k \in \mathbb{N}} \bigcup_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots t_n\}^k} B^{\mathcal{S}}_e\right) \\
&= \lim_{k \in \mathbb{N}} \mu_{\mathcal{S}}\left(\bigcup_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots t_n\}^k} B^{\mathcal{S}}_e\right) \quad \text{[Proposition 2.1.8]} \\
&= \lim_{k \in \mathbb{N}} \sum_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots t_n\}^k} \mu_{\mathcal{S}}(B^{\mathcal{S}}_e) \\
&= \lim_{k \in \mathbb{N}} \sum_{e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots t_n\}^k} \nu_{\mathcal{S}}(B^{\mathcal{S}}_e) \quad [\mu_{\mathcal{S}} \text{ extends } \nu_{\mathcal{S}}] \\
&= \lim_{k \in \mathbb{N}} \sum_{t'_1 \ldots t'_k \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots, t_n\}^k} \nu_{\mathcal{S}}(B^{\mathcal{S}}_{t'_1 \ldots t'_k}) \\
&= \lim_{k \in \mathbb{N}} \sum_{t'_1 \ldots t'_k \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots t_n\}^k} \prod_{1 \le i \le k} \mathrm{prob}_{\mathcal{S}}(t'_i) \\
&= \lim_{k \in \mathbb{N}} \sum_{t'_1 \ldots t'_k \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \cap \{t_1, \ldots t_n\}^k} \prod_{1 \le i \le k} \mathrm{prob}_{\mathcal{S}'}(t'_i) \\
&\qquad [\mathrm{prob}_{\mathcal{S}}(t_i) = \mathrm{prob}_{\mathcal{S}'}(t_i) \text{ for all } 1 \le i \le n] \\
&= \mu_{\mathcal{S}'}(\mathrm{Exec}^{\omega}_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^{\omega}) \quad \text{[by symmetric argument]}.
\end{aligned}
$$

From the above we can conclude that

$$
\begin{aligned}
&\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \Box p) \\
&= \mu_{\mathcal{S}}((\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) \cup (\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega)) \quad \text{[Theorem 4.3.17]} \\
&= \mu_{\mathcal{S}}(\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) + \mu_{\mathcal{S}}(\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega) \\
&= \mu_{\mathcal{S}'}(\{\, e(t_{\mathrm{target}_{\mathcal{S}'}(e)})^\omega \mid e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^* \,\}) + \mu_{\mathcal{S}'}(\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega) \\
&= \mu_{\mathcal{S}'}(\{\, e(t_{\mathrm{target}_{\mathcal{S}}(e)})^\omega \mid e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^* \,\} \cup (\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega)) \\
&= \mu_{\mathcal{S}'}(\{\, e' \in \mathrm{Exec}_{\mathcal{S}'} \mid e' \text{ does not contain } t_\perp \,\}).
\end{aligned}
$$

The last step follows from

$$
\begin{aligned}
&\{\, e' \in \mathrm{Exec}_{\mathcal{S}'} \mid e' \text{ does not contain } t_\perp \,\} \\
&\quad = \{\, \eta(e) \mid e \in (\mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^*) \cup (\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega) \,\} \\
&\quad = \{\, e(t_{\mathrm{target}_{\mathcal{S}}(e)})^\omega \mid e \in \mathrm{Exec}^*_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^* \,\} \cup (\mathrm{Exec}^\omega_{\mathcal{S}} \cap \{t_1, \ldots, t_n\}^\omega).
\end{aligned}
$$

□

The set of those execution paths of the searched system which contain the transition $t_\perp$ can be captured as a set of execution paths that reach the state $s_\perp$.

Corollary 4.4.6. *Assume that the search $t_1, \ldots, , t_n$ has not found a violation of $\Box p$.*

$$\mathrm{prog}_{\mathcal{S}}(t_1, \ldots, t_n, \Box p) = 1 - \mu_{\mathcal{S}'}(\{\, e' \in \mathrm{Exec}_{\mathcal{S}'} \mid e' \textit{ contains } t_\perp \}).$$

Several algorithms and tools are available to compute the probability of reaching a particular state (see, for example, [1, Section 10.1]).

4.5 Maintaining the Searched System

As we have seen in the previous section, to compute the progress measure of a search of a complete system for an invariant, we construct the corresponding searched system (and compute the measure of the set of those execution paths in the searched system that reach state $s_\perp$). As the search of the model checker continues, we would like to keep track of the progress. For a given complete system $\mathcal{S}$, this can be captured by the following diagram.

$$\begin{array}{ccc} t_1, \ldots, t_n & \longrightarrow & t_1, \ldots, t_n, t_{n+1} \\ \downarrow & & \downarrow \\ \mathcal{S}_n & & \mathcal{S}_{n+1} \end{array}$$

Rather than constructing the searched system from scratch after a new transition has been explored by the model checker, we show that we can construct the new searched system $\mathcal{S}_{n+1}$ from the old searched system $\mathcal{S}_n$ and the transition t_{n+1} in constant time.

But first we characterize the searched system corresponding to the empty search.

Proposition 4.5.1. *Let $\mathcal{S}_0$ be the searched system corresponding to the empty search. Then $S_0 = \{s_0, s_\perp\}$, $T_0 = \{t_{s_0}, t_\perp\}$, $AP_0 = AP_\mathcal{S} \cup \{\perp\}$, $\text{source}_0(t_{s_0}) = s_0$,*

$\text{source}_0(t_\perp) = s_\perp$, $\text{target}_0(t_\perp) = s_\perp$, $\text{prob}_0(t_{s_0}) = 1$, $\text{prob}_0(t_\perp) = 1$, $\text{label}_0(s_0) = \text{label}_{\mathcal{S}}(s_0)$ *and* $\text{label}_0(s_\perp) = \{\perp\}$. *If* s_0 *is final then* $\text{target}(t_{s_0}) = s_0$. *Otherwise,* $\text{target}(t_{s_0}) = s_\perp$.

Theorem 4.5.2. *Let* $\mathcal{S}_n$ *and* $\mathcal{S}_{n+1}$ *be the searched systems related to the searches* $t_1, \ldots, t_n$ *and* $t_1, \ldots, t_n, t_{n+1}$*, respectively. Let* $s_s = \text{source}(t_{n+1})$ *and* $s_t = \text{target}(t_{n+1})$. *Then*

$$S_{n+1} = S_n \cup \{s_s, s_t\}.$$

and

$$T_{n+1} = \begin{cases} T_n \cup \{t_{n+1}\} \cup \{t_{s_s}\} \cup \{t_{s_t}\} & \text{if } s_s \notin S_n \wedge s_t \notin S_n \wedge \text{prob}_{\mathcal{S}}(t_{n+1}) < 1 \\ T_n \cup \{t_{n+1}\} \cup \{t_{s_t}\} & \text{if } s_s \notin S_n \wedge s_t \notin S_n \wedge \text{prob}_{\mathcal{S}}(t_{n+1}) = 1 \\ T_n \cup \{t_{n+1}\} \cup \{t_{s_s}\} & \text{if } s_s \notin S_n \wedge s_t \in S_n \wedge \text{prob}_{\mathcal{S}}(t_{n+1}) < 1 \\ T_n \cup \{t_{n+1}\} & \text{if } s_s \notin S_n \wedge s_t \in S_n \wedge \text{prob}_{\mathcal{S}}(t_{n+1}) = 1 \\ T_n \cup \{t_{n+1}\} \cup \{t_{s_t}\} & \text{if } s_s \in S_n \wedge s_t \notin S_n \wedge \text{prob}_n(t_{s_s}) > \text{prob}_{\mathcal{S}}(t_{n+1}) \\ T_n \cup \{t_{n+1}\} \cup \{t_{s_t}\} \setminus \{t_{s_s}\} & \text{if } s_s \in S_n \wedge s_t \notin S_n \wedge \text{prob}_n(t_{s_s}) = \text{prob}_{\mathcal{S}}(t_{n+1}) \\ T_n \cup \{t_{n+1}\} & \text{if } s_s \in S_n \wedge s_t \in S_n \wedge \text{prob}_n(t_{s_s}) > \text{prob}_{\mathcal{S}}(t_{n+1}) \\ T_n \cup \{t_{n+1}\} \setminus \{t_{s_s}\} & \text{if } s_s \in S_n \wedge s_t \in S_n \wedge \text{prob}_n(t_{s_s}) = \text{prob}_{\mathcal{S}}(t_{n+1}) \end{cases}$$

and

$$AP_{n+1} = AP_n$$

and the function $\text{source}_{n+1} : T_{n+1} \to S_{n+1}$ *satisfies*

$$\text{source}_{n+1}(t) = \begin{cases} s_s & \textit{if } t = t_{n+1} \\ s_s & \textit{if } t = t_{s_s} \\ s_t & \textit{if } t = t_{s_t} \\ \text{source}_n(t) & \textit{otherwise} \end{cases}$$

and the function $\text{target}_{n+1} : T_{n+1} \to S_{n+1}$ *satisfies*

$$\text{target}_{n+1}(t) = \begin{cases} s_t & \textit{if } t = t_{n+1} \\ s_\perp & \textit{if } t = t_{s_s} \\ s_t & \textit{if } t = t_{s_t} \wedge s_t \textit{ is final} \\ s_\perp & \textit{if } t = t_{s_t} \wedge s_t \textit{ is partial} \\ \text{target}_n(t) & \textit{otherwise} \end{cases}$$

and the function $\text{prob}_{n+1} : T_{n+1} \to (0, 1]$ *satisfies*

$$\text{prob}_{n+1}(t) = \begin{cases} \text{prob}_{\mathcal{S}}(t_{n+1}) & \textit{if } t = t_{n+1} \\ \text{prob}_n(t_{s_s}) - \text{prob}_{\mathcal{S}}(t_{n+1}) & \textit{if } t = t_{s_s} \wedge s_s \in S_n \wedge \text{prob}_n(t_{s_s}) > \text{prob}_{\mathcal{S}}(t_{n+1}) \\ 1 - \text{prob}_{\mathcal{S}}(t_{n+1}) & \textit{if } t = t_{s_s} \wedge s_s \notin S_n \wedge \text{prob}_{\mathcal{S}}(t_{n+1}) < 1 \\ 1 & \textit{if } t = t_{s_t} \wedge s_t \notin S_n \\ \text{prob}_n(t) & \textit{otherwise} \end{cases}$$

and the function $\text{label}_{n+1} : S_{n+1} \to 2^{AP_{n+1}}$ *satisfies*

$$\text{label}_{n+1}(s) = \begin{cases} \text{label}_n(s) & \textit{if } s \in S_n \\ \text{label}_{\mathcal{S}}(s) & \textit{otherwise} \end{cases}$$

Algorithm 1 Implementation of Proposition 4.5.1 and Theorem 4.5.2

Initialize()

$S \leftarrow$ empty set

$T \leftarrow$ empty set

$\text{insert}(S, s_0); \text{insert}(S, s_\perp)$

$\text{source}(t_\perp) \leftarrow s_\perp; \text{target}(t_\perp) \leftarrow s_\perp; \text{prob}(t_\perp) \leftarrow 1; \text{insert}(T, t_\perp)$

if s_0 is final **then**

    $\text{target}(t_{s_0}) \leftarrow s_0$

else

    $\text{target}(t_{s_0}) \leftarrow s_\perp$

$\text{source}(t_{s_0}) \leftarrow s_0; \text{prob}(t_{s_0}) \leftarrow 1; \text{insert}(T, t_{s_0})$

Require: $t \notin T$

Add(t)

$s_s \leftarrow \text{source}(t); s_t \leftarrow \text{target}(t)$

$\text{insert}(T, t)$

if $s_s \notin S$ **then**

    $\text{insert}(S, s_s)$

    if $\text{prob}(t) < 1$ **then**

        $\text{source}(t_{s_s}) \leftarrow s_s; \text{target}(t_{s_s}) \leftarrow s_\perp; \text{prob}(t_{s_s}) \leftarrow 1 - \text{prob}(t); \text{insert}(T, t_{s_s})$

else

    if $\text{prob}(t) < \text{prob}(t_{s_s})$ **then**

        $\text{prob}(t_{s_s}) \leftarrow \text{prob}(t_{s_s}) - \text{prob}(t)$

    else

The above results provide the correctness proof of Algorithm 1.

Proposition 4.5.3. *The worst-case running time of Initialize is constant. The amortized expected running time per Add is constant.*

Proof. The only operations which may not be constant time are the tests in line 15 and 24, the inserts in line 16, 18 and 30, the look up of t_{s_s} in line 20, and the remove in line 23. By exploiting dynamic perfect hashing, as discussed in [6], the amortized expected running time can be shown to be constant. □

5 Search Strategies

Explicit-state model checkers like JPF use search strategies such as depth-first search (DFS) and breadth-first search (BFS) to traverse the state space of the system under verification. First, we present the traditional search strategies DFS and BFS within the model we presented in Chapter 3. Next, we present three new search strategies. In contrast to DFS and BFS, these search strategies take the probabilities associated with the random choices into account. As we shall see in Chapter 9, these search strategies generally make progress faster than the traditional search strategies DFS and BFS.

5.1 DFS and BFS

Given a probabilistic transition system, a search strategy visits the states and the transitions of the system in a systematic way. We are mainly focused on the order in which the transitions are visited. Hence, a search strategy gives rise to a sequence of transitions.

In our implementation of DFS we use a stack to store transitions. An algorithm for DFS can be found in Algorithm 2.

Algorithm 2 Depth-first search

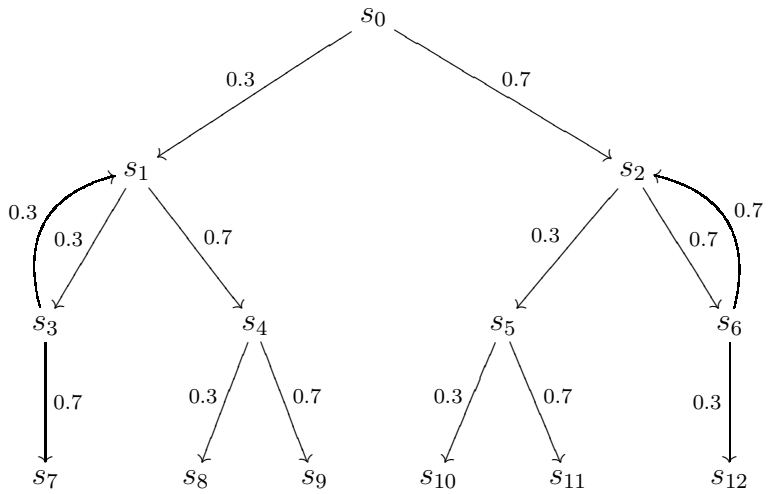
Require: no state is marked visited

1: $st \leftarrow$ empty stack
2: **for all** transitions t from s_0 **do**
3: push(st, t)
4: mark s_0 visited
5: **while** st is nonempty **do**
6: $t \leftarrow$ pop(st) ▷ t is visited
7: **if** target(t) is not visited **then**
8: **for all** transitions t' such that source(t')=target(t) **do**
9: push(st, t')
10: mark target(t) visited

In [18], Knuth and Yao show how a die can be implemented by means of a coin. We use a biased version, presented in the next example, as our running example in this chapter.

Example 5.1.1. An implementation of a biased die by means of a biased coin is

represented by the probabilistic transition system depicted below.



For transitions, we use the same naming convention as in Example 3.1.2. Sometimes we add a - to enhance readability, as for example in t_{5-10}.

A DFS can visit the transitions in the following order: $t_{01}, t_{13}, t_{31}, t_{37}, t_{14}, t_{48}, t_{49}, t_{02}, t_{25}, t_{5-10}, t_{5-11}, t_{26}, t_{6-12}, t_{62}$

Instead of a stack, BFS uses a queue to store transitions. An algorithm for BFS can be found in Algorithm 3. For the probabilistic transition system of Example 5.1.1, BFS can visit the transitions in the following order: $t_{01}, t_{02}, t_{13}, t_{14}, t_{25}, t_{26}, t_{31}, t_{37}, t_{48}, t_{49}, t_{5-10}, t_{5-11}, t_{6-12}, t_{62}$.

5.2 Probability-First Search

Search strategies such as DFS and BFS do not take the probability of transitions into account. To let transitions with the highest probability be searched first,

Algorithm 3 Breadth-first search

Require: no state is marked visited

```
1: q ← empty queue
2: for all transitions t from s_0 do
3:     enqueue(q, t)
4: mark s_0 visited
5: while q is nonempty do
6:     t ← dequeue(q)                                  ▷ t is visited
7:     if target(t) is not visited then
8:         for all transitions t' from target(t) do
9:             enqueue(q, t')
10:        mark target(t) visited
```

our probability-first search (PFS) strategy sorts the enabled transitions by their probability. This search strategy is similar to A*[13]. For the probabilistic transition system of Example 5.1.1, PFS visits the transitions in the following order: $t_{02}, t_{26}, t_{62}, t_{01}, t_{25}, t_{14}, t_{6-12}, t_{5-11}, t_{49}, t_{13}, t_{5-10}, t_{48}, t_{37}, t_{31}$.

To implement PFS, we exploit a priority queue. Each key is a real number, which represents a probability. The keys are ordered as follows:

$$p_1 \preceq p_2 \text{ if } p_1 \geq p_2.$$

The elements of the priority queue are transitions. An algorithm for PFS can be found in Algorithm 4.

5.3 Breadth-First Probability-Second Search

Our breadth-first probability-second search (BFPSS) is an enhancement of BFS in which transitions at the same level are sorted by their probability. For the probabilistic transition system of Example 5.1.1, BFPSS visits the transitions in the following order: $t_{02}, t_{01}, t_{26}, t_{14}, t_{25}, t_{13}, t_{62}, t_{5-11}, t_{49}, t_{37}, t_{6-12}, t_{5-10}, t_{48}, t_{31}$

To implement BFPSS, we also exploit a priority queue. This time, each key is a pair consisting of an integer, which represents a level, and a real, which represents a probability. The keys are ordered as follows:

$$[\ell_1, p_1] \preceq [\ell_2, p_2] \text{ if } \ell_1 < \ell_2 \vee (\ell_1 = \ell_2 \wedge p_1 \geq p_2).$$

Algorithm 4 Probability-first search

Require: no state is marked visited

1: $q \leftarrow$ empty priority queue
2: **for all** transitions t from s_0 **do**
3: insert$(q, \langle \text{prob}(t), t \rangle)$
4: mark s_0 visited
5: **while** q is nonempty **do**
6: $\langle p, t \rangle \leftarrow \text{deleteMin}(q)$ ▷ t is visited
7: **if** target(t) is not visited **then**
8: **for all** transitions t' from target(t) **do**
9: enqueue$(q, \langle \text{prob}(t') \times p, t' \rangle)$
10: mark target(t) visited

The elements of the priority queue are transitions. An algorithm for BFPSS can be found in Algorithm 5.

Algorithm 5 Breadth-first probability-second search

Require: no state is marked visited

```
1:  q ← empty priority queue
2:  for all transitions t from s_0 do
3:      insert(q, ⟨[1, prob(t)], t⟩)
4:  mark s_0 visited
5:  while q is nonempty do
6:      ⟨[ℓ, −], t⟩ ← deleteMin(q)                    ▷ t is visited
7:      if target(t) is not visited then
8:          for all transitions t′ from target(t) do
9:              insert(q, ⟨[ℓ + 1, prob(t′)], t′⟩)
10:         mark target(t) visited
```

5.4 Randomized Search

Our randomized search (RS) randomly selects an enabled transition. The chance that a transition is selected is proportional to its probability.

To implement RS, we use a set. The elements of the set are pairs, each consisting of a real, representing a probability, and a transition. The operation select(s)

removes an element $\langle p, t \rangle$ from the set and returns the element. The probability that the element $\langle p, t \rangle$ is selected is

$$\frac{p}{\sum_{\langle p', t' \rangle \in s} p'}.$$

An algorithm for RS is given in Algorithm 6.

Algorithm 6 Randomized search

Require: no state is marked visited

$s \leftarrow$ empty set
for all transitions t from s_0 **do**
    add$(s, \langle \text{prob}(t), t \rangle)$
mark s_0 visited
while s is nonempty **do**
    $\langle p, t \rangle \leftarrow$ select(s) ▷ t is visited
    if target(t) is not visited **then**
        for all transitions t' from target(t) **do**
            add$(s, \langle \text{prob}(t') \times p, t' \rangle)$
        mark target(t) visited

5.5 Properties of Search Strategies

First, we present a property of the algorithms that we shall use in several proofs. In this section, we use T_0 to denote the set of those transitions that can be reached from s_0.

Proposition 5.5.1. *All transitions added to the collection belong to T_0.*

Proof. We show that the fact that all transitions added to the collection belong to T_0 is a loop invariant of the loop in line 5-10. Obviously, the transitions added to the collection in line 3 are reachable from s_0 and, hence, belong to T_0.

Next we show that this loop invariant is maintained by the loop in line 5-10. By the loop invariant we know that t belongs to T_0, i.e. t is reachable from s_0. Since $\text{target}(t) = \text{source}(t')$, t' is also reachable from s_0 and, hence, belongs to T_0. □

Like DFS and BFS, PFS, BFPSS and RS visit each transition at most once.

Proposition 5.5.2. *In PFS, BFPSS and RS, each transition is visited at most once. If T_0 is finite then PFS, BFPSS, and RS visit all transitions in T_0.*

Proof. First, we prove that each transition is visited at most once.

- By inspecting the algorithms, we know once a state is marked visited, it remains marked visited.

- From line 4 and line 10 of these algorithms, we know that after a transition t has been added to the collection, which is a priority-queue for PFS and BFPSS and a set for RS, source(t) is marked visited.

- From the precondition and line 7, we know that at the time a transition t is added to the collection, source(t) is not marked visited.

So each transition t is added to the collection at most once. From this and line 6 of the algorithms, where a transition is visited and removed from the collection, we can conclude that each transition is visited at most once.

Next, we prove that all transitions in T_0 will be visited by contradiction. Let us assume there exists some transition t in T_0 that is not visited by the search strategies. Because t is in T_0, we know that t can be reached from s_0 via a prefix of an execution path. Say e is that prefix from s_0 to source(t). Let t' be the first transition in et that has not been visited by the search strategies. Such a t' exists because t has not been visited.

Now we distinguish two cases.

- Assume that source(t') $= s_0$. In line 2 and 3, we have added all outgoing transitions of s_0 and hence t' to the collection. From Proposition 5.5.1 we know that all the transitions added to the collection belong to T_0. Since also T_0 is finite, we can conclude that t' is removed from the collection and hence

visited.

- Assume that source(t') $\neq s_0$. Let e' be such that $e't'$ is a prefix of et. Since t' is the first transition that has not been visited, all transitions in e' have been visited. Let t'' be the last transition of e'. Initially, state target(t'') is not marked visited. Consider the first time line 7 is executed for state target(t''). (Note that line 7 is executed at least once for state target(t'') because t'' has been visited.) Then in line 9, transition t' is added to the collection. The remainder of the proof is the same as in the previous case.

□

Since PFS, BFPSS and RS take the probabilities into account, these search strategies are not as efficient as DFS and BFS.

Proposition 5.5.3. *If T_0 is finite and*

- *a state can be marked visited in $O(1)$ time,*
- *a state can be checked to be marked in $O(1)$ time,*
- *the outgoing transitions of a state s can be enumerated in $O(n)$ time, where n is the number of outgoing transitions of s,*

then the worst-case running time of PFS, BFPSS, and RS is $O(|T_0| \log |T_0|)$.

Proof. We represent the priority queue by means of a heap. Therefore, the worst-case running time of insert and deleteMin are $O(\log(n))$, where n is the size of the priority queue. We represent the set by means of an augmented red-black tree [4, Chapter 14]. Therefore, the worst-case running time of adding and selecting are $O(\log n)$, where n is the size of the set. From the RS algorithm we can deduce that whenever we add a transition to the set, that transition is not already in the set. Hence, the running time of add is $O(1)$. The worst-case running time of select is $O(n)$, where n is the size of the set.

- From Proposition 5.5.2, we know every transition in T_0 will be visited once and, hence, will be added to the collection once. So line 2–3 and line 8–9 will be executed $|T_0|$ times. From our assumptions we can conclude that the running time for all the executions of these lines combined is $O(|T_0| \log |T_0|)$.

- Line 6 is executed $|T_0|$ times. In the worst case, there are $|T_0|$ transitions in the collection. Thus, the worst-case running time for all the executions of line 6 combined is $O(|T_0| \log |T_0|)$.

- Line 5 and 7 are executed $|T_0|$ times. Hence, in total they contribute $O(|T_0|)$ to the running time.

- Line 10 is executed at most $|T_0|$ times. Hence, in total it contributes $O(|T_0|)$ to the worst-case running time.

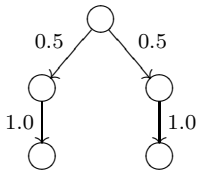
- Line 1 and 4 are executed once. Hence, in total they contribute $O(1)$ to the running time.

Adding the above leads us to the desired results. □

5.6 Comparison

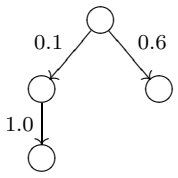
Our progress measure allows us to compare the amount of progress these different search strategies make. Next we shall provide examples that show that the four different search strategies of DFS, BFS, PFS, and BFPSS are incomparable. That is, for each pair of search strategies, we shall construct a complete system such that one strategy makes faster progress than the other.

The first example shows that DFS can make faster progress than BFS.



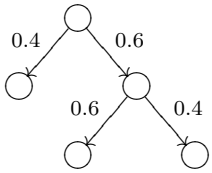
DFS	BFS
0	0
0.5	0
0.5	0.5
1.0	1.0

The second example shows the opposite, that is, BFS can make faster progress than DFS.



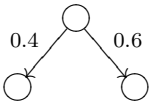
BFS	DFS
0	0
0.6	0.4
1.0	1.0

Our third example shows that both DFS and BFS can make progress faster than PFS and BFPSS.



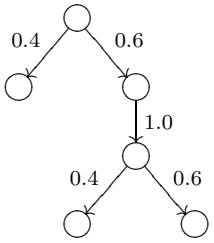
DFS	BFS	PFS	BFPSS
0.4	0.4	0	0
0.4	0.4	0.4	0.4
0.76	0.76	0.76	0.76
1.0	1.0	1.0	1.0

The fourth example shows the opposite, that is, both PFS and BFPSS can make faster progress than DFS and BFS.



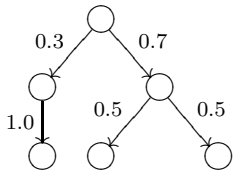
PFS	BFPSS	DFS	BFS
0.6	0.6	0.4	0.4
1.0	1.0	1.0	1.0

Next, we present an example of a system for which BFPSS makes faster progress than PFS.



BFPSS	PFS
0	0
0.4	0
0.4	0.4
0.76	0.76
1.0	1.0

Finally, we show that PFS can make progress faster than BFPSS.



PFS	BFPSS
0	0
0.35	0
0.7	0.3
0.7	0.65
1.0	1.0

As we have seen, DFS, BFS, PFS and BFPSS are in general incomparable. Since RS can behave as any of them, RS is incomparable to the other search strategies as well. In Chapter 9 we shall compare the progress made by these search strategies for a number of randomized algorithms.

6 An Extension of JPF to a Probabilistic Model Checker

We are interested in model checking randomized algorithms implemented in Java using JPF. These randomized algorithms contain at least one randomized choice. JPF does not know about the probabilities associated with the randomized choices. However, JPF needs this information to, for example, keep track of its progress (the need for a progress measure was already discussed earlier). To accomplish this, we have to introduce several new classes and interfaces.

6.1 The Class Choice

One way to express a randomized choice in Java is to use the `nextInt(int n)` method of the `Random` class. This method returns the integer i with probability $\frac{1}{\mathtt{n}}$ where $0 \leq i < \mathtt{n}$. For example, consider a randomized version of Hoare's quicksort [14]. This algorithm only contains a single randomized choice which is used to randomly select a pivot to split the list. This randomized algorithm can be

implemented in Java as follows.

```
1  /**
2      Sorts the element of the given list.
3      @param list the list to be sorted.
4      @pre. all elements of the list are distinct and different from null
           .
5   */
6  public static void quickSort(ArrayList<T> list) {
7      if (list.size() != 0) {
8          //select a random pivot
9          Random random = new Random();
10         int index = random.nextInt(list.size());
11         T pivot = list.get(index);
12         //partition list into two
13         ArrayList<T> small = new ArrayList<T>(list.size());
14         ArrayList<T> large = new ArrayList<T>(list.size());
15         for (T element : list) {
16             if (element.compareTo(pivot) < 0) {
17                 small.add(element);
18             } else if (element.compareTo(pivot) > 0) {
```

```
19                  large.add(element);
20              }
21          }
22          //recursively sort
23          quickSort(small);
24          quickSort(large);
25          //combine the sorted list
26          list.clear();
27          list.addAll(small);
28          list.add(pivot);
29          list.addAll(large);
30      }
31  }
```

However, there are several other ways to implement randomized choices in Java. For example, the above randomized choice can also be implemented as follows by using the `nextDouble` method of the `Random` class.

```
int index = (int) (random.nextDouble() * input.size());
```

In the first approach, an invocation of the `nextInt` method represents a randomized choice. Such an invocation can easily be detected by JPF. In the second approach,

a randomized choice is not just represented by an invocation of the `nextDouble` method. In this case, it is a particular combination of the `nextDouble` method, the `size` method, multiplication and casting. Detecting such a combination is much more difficult in JPF.

Also in some cases we may need other distributions than the uniform distribution. For instance, when we want to simulate a biased coin, we may want to specify the probability of heads to be 0.7. This cannot be directly expressed using the `nextInt` method.

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

An invocation of the `make` method contains the probabilities of the randomized choice. In our implementation of quicksort we first have to create an array `p` of doubles, each of which has the value $\frac{1.0}{\texttt{list.size()}}$, and use the `make` method to make the choice as follows.

```
1 double p[] = new double[list.size()];
2 for (int i = 0; i <  list.size(); i ++) {
3     p[i] = 1.0 / list.size();
```

```
4 }
5 int index = Choice.make(p);
```

We implement of the `Choice` class as follows

```
1  package probabilistic;
2  public class Choice {
3    /**
4       Given an array p of probabilities, returns i with probability p[i
            ], where 0 <= i < p.length.
5       @param p probabilities.
6       @pre. p[0] + ... + p[p.length - 1] = 1.0.
7       @return i with probability p[i].
8     */
9    public static int make (double[] p) {
10     double sum = p[0];
11     double choice = Math.random();
12     int alternative = 0;
13     while (choice >= sum) {
14       alternative++;
15       sum += p[alternative];
```

```
16      }
17      return alternative;
18    }
19  }
```

6.2 The Abstract Class ChoiceGenerator

Java programs give rise to different types of choices. For example, methods `nextInt` and `nextDouble` give rise to randomized choices. And concurrent Java programs give rise to nondeterministic choices due to the interleaving of different threads. JPF needs to represent these choices. Therefore the class `ChoiceGenerator` has been introduced.

Note that the class `ChoiceGenerator` is abstract. This provides us with flexibility because it allows us to extend JPF so that it can handle randomized choices without having to change any of the core classes of JPF. The only thing we need to do is to develop a concrete class to extend the abstract class `ChoiceGenerator`. This concrete class then can be used by JPF.

There are several subclasses of `ChoiceGenerator` in JPF.

- The subclass `IntIntervalGenerator` represents a choice among the integers within a given interval. The bounds of the interval are given as arguments to

the constructor.

- The subclass `DoubleChoiceFromSet` represents a choice among a set of doubles. The set of doubles is specified in a configuration file.

- The subclass `ThreadChoiceFromSet` represents a choice among a number of threads.

Each choice consists of a number of alternatives of which one can be chosen. In order to enumerate these alternatives, the class `ChoiceGenerator` contains the following three abstract methods.

- The method `getTotalNumberOfChoices` returns the total number of alternatives of this choice.

- The method `getNextChoice` returns the next alternative of this choice.

- The method `hasMoreChoices` tests whether there are more alternatives of this choice.

In Section 6.4 we shall introduce the class `ProbabilisticChoiceGenerator`, which extends the class `ChoiceGenerator`, to represent randomized choices.

6.3 The Interface Probable

To make it easy to extend JPF in the future with others ways to represent randomized choices, we introduce the interface `Probable`. The interface contains the abstract method `getProbability`. During model checking, after a randomized choice has been made, we can invoke this method to retrieve the probability associated with the alternative of the choice being checked by JPF.

```
1  package gov.nasa.jpf.jvm.choice;
2  /**
3     Objects of this type have a probability.
4   */
5  public interface Probable {
6    /**
7       Returns a probability.
8       @return a probability.
9     */
10   public double getProbability();
11 }
```

The interface `Probable` provides flexibility. By using this interface we do not need to specify which class we need when we want to extract the probability of the

current alternative of a choice. It allow us to add another class which represents randomized choices without modifying our extension of JPF. For example, assume that we want to extend JPF so that it can handle randomized choices expressed by means of the method `Coin.flip`. This method returns true when the coin flip results in heads and false otherwise. In such a case, we introduce a new class to represent the choice among the alternatives of heads and tails. This class implements the interface `Probable` and, hence, contains the method `getProbability` which returns 0.5 for each alternative.

6.4 The Class ProbabilisticChoiceGenerator

Since we need to keep track of the probabilities associated with randomized choices, we introduce the class `ProbabilisticChoiceGenerator`. This class is an extension of the class `IntIntervalGenerator`, which implements the interface `ChoiceGenerator`. JPF can use the `ProbabilisticChoiceGenerator` object to enumerate the alternatives of a randomized choice. The class `ProbabilisticChoiceGenerator` implements the interface `Probable`. Thus the class `ProbabilisticChoiceGenerator` needs to provide the probability of the current alternative of a randomized choice.

The class `ProbabilisticChoiceGenerator` has only one constructor. This constructor takes an array of doubles as its argument. This array `p` represents the probabilities of the alternatives of a randomized choice. In the constructor,

- we invoke the constructor of the super class `IntIntervalGenerator` with the arguments 0 and `p.length-1`, which represents a choice among integers in the interval $[0, \texttt{p.length} - 1]$, and
- we keep a copy of the array `p` of probabilities so that even if the array is modified by JPF later, it will not affect this randomized choice.

Whenever a JPF component invokes the method `getProbability` of a `ProbabilisticChoiceGenerator` object, it returns the probability of the current alternative of this choice.

6.5 The Native Peer Class JPF_probabilistic_Choice

JPF does not know about the probabilities associated with the randomized choices expressed by means of the `make` method. However, JPF needs this information, for example, to keep track of its progress. In order to extract this information, we introduce a so-called native peer class named `JPF_probabilistic_Choice`. The class `JPF_probabilistic_Choice` is named according to JPF's convention for naming native peer classes. Like the `Choice` class, this class also contains the method `make`. Whenever JPF encounters an invocation of the `probabilistic.Choice.make` method, it does not model check that method, but it executes (but not model checks) the `JPF_probabilistic_Choice.make` method instead. The latter method provides

JPF with the probabilities associated with the randomized choice represented by the `make` method.

Each method in a native peer class has two additional parameters, hence the `make` method of class `JPF_probabilistic_Choice` has three parameters.

- The first parameter is a `MJIEnv` object. Using this object, we can retrieve information of JPF, such as thread, state, and memory information.
- The second parameter is an integer. This is the integer used by JPF to represent the entity on which the method is invoked. In our case it is the `Choice` class. However, since we do not use it in the `make` method, we call it `dummy`.
- The third parameter is an integer used by JPF to represent the parameter of the `make` method, that is, the array of doubles. We use this integer to retrieve the array through the `MJIEnv` object.

The `JPF_probabilistic_Choice.make` method returns an integer, which corresponds to an alternative of the choice represented by the method `Choice.make`.

Its code is shown below.

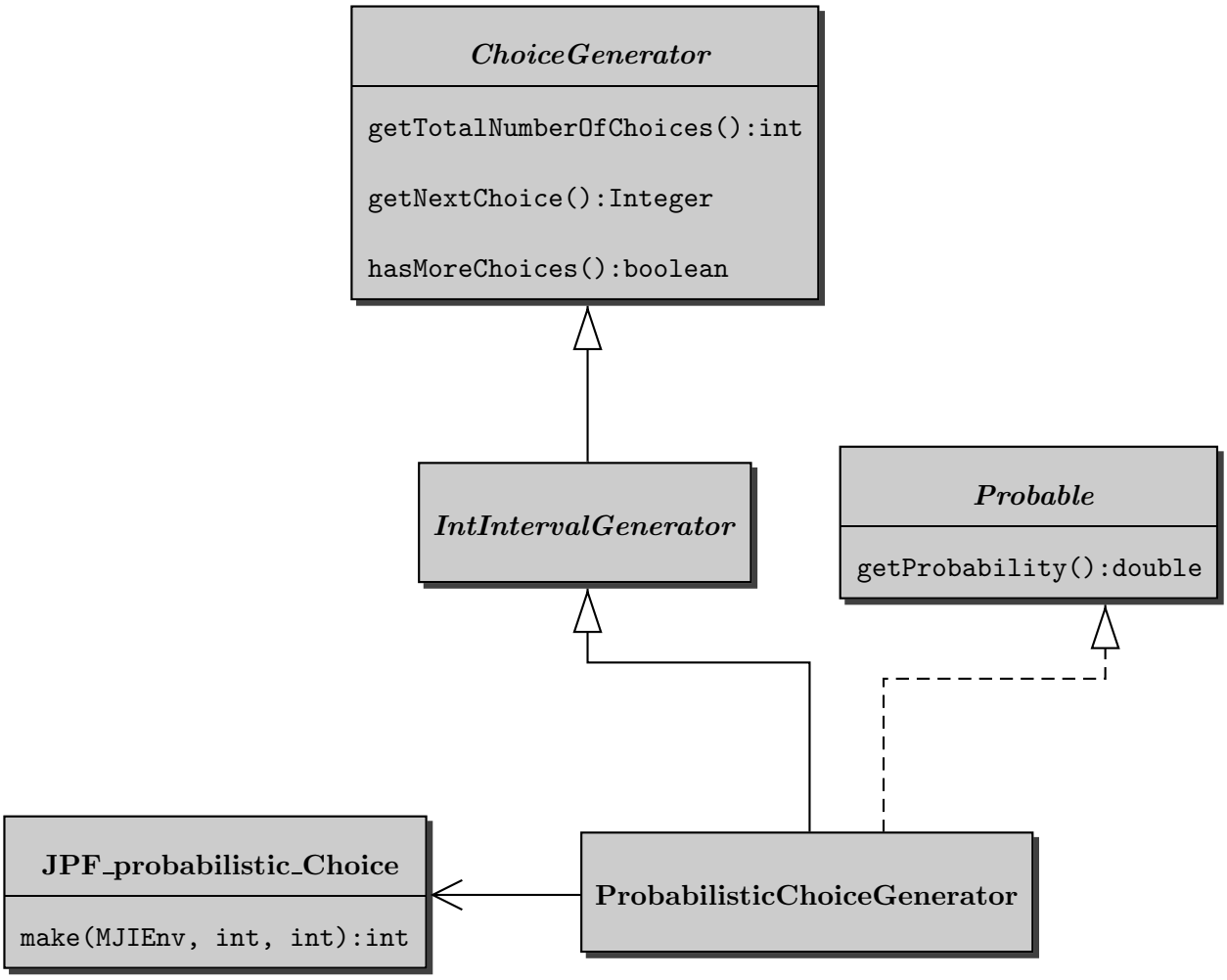
```
1 /**
2     This class is the native peer of the class probabilistic.Choice.
3     @see probabilistic.Choice
```

```
4  */
5  public class JPF_probabilistic_Choice {
6    /**
7       Returns the number of times this invocation of the make method has
            been encountered before to JPF.  This number corresponds to a
           choice. The first return is ignored by JPF.
8       @param env JPF environment.
9       @param dummy arbitrary integer (plays no role in the method, but
            this parameter is needed for JPF to work properly).
10      @param pRef reference to the array p of probabilities.
11      @pre. p[0] + ... + p[p.length - 1] = 1.0.
12      @return the number of times this invocation of the make method has
            been encountered before.
13     */
14   public static int make(MJIEnv env, int dummy, int pRef) {
15     ThreadInfo ti = env.getThreadInfo();
16     SystemState ss = env.getSystemState();
17     ChoiceGenerator cg;
18     if (!ti.isFirstStepInsn()) {
19       double[] p = env.getDoubleArrayObject(pRef);
```

```
20          cg = new ProbabilisticChoiceGenerator(p);
21          ss.setNextChoiceGenerator(cg);
22          env.repeatInvocation();
23          return -1; // JPF ignores this return
24        } else {
25          cg = ss.getChoiceGenerator();
26          return ((ProbabilisticChoiceGenerator) cg).getNextChoice();
27        }
28      }
29  }
```

6.6 The Complete Picture

In the previous sections we have discussed all the classes and interfaces needed for JPF to manipulate the randomized choices and to extract probabilities of these choices. The UML diagram below shows the relationships among them.



When we are using JPF to model check the following simple example, we can use our classes and interfaces to enumerate the alternatives of a randomized choice.

```
1 import probabilistic.*;
2 public class SimpleExample {
```

```
3       public static void main(String[] args) {
4           long counter = 0;
5           while (Choice.make(0.5,0.5) == 1) {
6             counter++;
7           }
8       }
9   }
```

When JPF encounters the method invocation `Choice.make(0.5,0.5)`, since we introduced the native peer class `JPF_probabilistic_Choice`, instead of model checking the method `Choice.make`, JPF executes the static method `make` of the class `JPF_probabilistic_Choice`. Within this method we create an object of type `ProbabilisticChoiceGenerator` and use this object to manipulate the two alternatives of the choice. And since the class `ProbabilisticChoiceGenerator` implements the interface `Probable`, we can retrieve the probability 0.5 for each alternative of the choice by means of the method `getProbability`.

In future chapters we shall discuss how the probability of randomized choices can be retrieved and how this probability can be used.

7 Implementation of the Progress Measure

As we showed in Chapter 6, we have extended the model checker JPF to a probabilistic model checker. In Section 4.2, we introduced a notion of progress and in Section 4.3 we gave a characterization of progress for invariants. In this chapter we will discuss how we can implement the algorithm of Section 4.5 to compute progress for invariants in JPF.

JPF is an event-driven system. In such a system, one can associate a so-called event with a particular condition. For example, in JPF an event is associated with the start of the model checking. When JPF starts its model checking, this event is triggered. In an event-driven system, the triggering of an event may give rise to the execution of some code by a so-called listener. Listeners can be associated, also known as registered, to events at runtime. In JPF, the listeners are usually specified in the properties file `jpf.properties`. For example, all registered listeners are notified when JPF starts its search. In JPF, the code associated to an event is defined in a method. For example, the code fragment triggered by the event

associated with the start of JPF is defined in a method named `searchStarted`. When JPF starts its search, the method `searchStarted` of all registered listeners is invoked. For more information on event-driven programming, we refer readers to [11].

To implement the computation of the progress measure for invariants, as we discussed in Chapter 4, we exploit event-driven programming. As we shall see, by implementing a new listener, which we call `ProbabilityListener`, and by registering this new listener in the file `jpf.properties`, we can extend JPF so that it keeps track of its progress without modifying its core. This listener extracts information from JPF, such as the current state, the choice being made, the current alternative of the choice, the probability of the alternative and so on. It uses this information and the algorithms presented in Section 4.5 to compute progress. Since the algorithms make use of probabilistic transition systems, we introduce the classes `PTS` and `Transition` to represent probabilistic transition systems and their transitions, respectively. As we have seen in Section 4.3, the progress of invariants can be characterized as the probability of reaching a particular state in a probabilistic transition system. To compute that probability, we use the tool MRMC. MRMC is a model checker developed by the Software Modeling and Verification (MOVES) group at RWTH Aachen [17].

7.1 The Class ProbabilityListener

During model checking, JPF notifies the registered listeners when an event is triggered. For example, when JPF finds a new state, it will trigger a `stateAdvanced` event and a listener can act accordingly. There are three types of listeners in JPF. Each type of listener is represented as an interface.

- `SearchListener`. A `SearchListener` is notified by a search strategy. For example, when a search strategy starts, JPF notifies each registered `SearchListener` of the event of `searchStarted`, that is, it invokes the `searchStarted` method of each registered `SearchListener` object.

- `VMListener`. JPF notifies each registered `VMListener` when the JVM triggers certain events. For example, when the JVM executes an instruction, JPF notifies each registered `VMListener` of the event of `instructionExecuted`, that is, it invokes the `instructionExecuted` method of each registered `VMListener` object.

- `PublisherExtension`. A registered `PublisherExtension` is notified when JPF reports the final results, such as whether there is violation of a property, and statistics of memory usage. For example, when JPF finishes model checking Java bytecode, JPF notifies each registered `PublisherExtension` of the event of `publishFinished`, that is, it invokes the `publishFinished` method

of each registered `PublisherExtension` object.

JPF does not only have interfaces, but also provides a `ListenerAdapter` class, which implements all the three interfaces with empty methods. So instead of implementing the interfaces directly and providing an implementation for all the abstract methods defined in the interfaces, we can simply override a method of the class `ListenerAdapter`.

Our class `ProbabilityListener` extends the class `ListenerAdapter`. We use this class to compute the progress measure which we defined in Chapter 4. In our class we have overridden the following methods.

- `searchStarted`. This method is invoked when JPF starts a search. In this event we create a new probabilistic transition system as we defined in Proposition 4.5.1. We also retrieve information such as `probability.maximum.transitions`, which allows us to specify the maximum number of transitions to be verified, from the configuration file.

- `stateAdvanced`. This method is invoked when JPF executes an instruction and finds a new state. Since in this event a new state is found, we add a new transition to the existing probabilistic transition system according to Theorem 4.5.2. Here we compute the progress measure if the number of transitions added to the probabilistic transition system is a multiple of

`probability.measure.sampling_interval`, which is specified in the configuration file. When JPF reaches the number of transitions specified by `probability.maximum.transitions` in the configuration file, we force JPF to quit the model checking.

- `stateBacktracked` and `stateRestored`. These methods are invoked when JPF moves to a previously visited state. In such an event we reset the current state to the state to which JPF moved. This state may be needed as the source of a new transition.

- `searchFinished`. This method is invoked when JPF finishes the search. In this event we free the resources. We shall provide more details in Section 7.4.

As we discussed, we need to represent a probabilistic transition system including its transitions. Therefore, we introduce the classes `Transition` and `PTS` in the next two sections.

7.2 The Class Transition

We use the class `Transition` to represent a transition of a probabilistic transition system. In this class, we represent the state of a probabilistic transition system by an integer.

In a probabilistic transition system, a transition does not only include the

source and target state, but also the probability of the transition. Thus in our class `Transition` we have the attributes `source` and `target` of type `int`, and `probability` of type `double`.

Furthermore, since in the construction of the complete system (see Definition 4.4.1) we need to know whether a state is final, we introduce an attribute `isTargetFinal` of the type of `boolean` as well.

These four attributes are not sufficient to identify transitions in a probabilistic transition system generated by JPF, since in JPF there may be more than one transition with the same `source`, `target` and `probability`. Therefore, we introduce an attribute `id` in the class `Transition`. The `id` represents the id of an alternative of a randomized choice.

7.3 The Class PTS

As we have seen in Section 4.4, we use a probabilistic transition system to compute the progress. We use the class `PTS` to represent a probabilistic transition system. This class contains the following data.

- We use a set of `Integer`s to represent the set of states.
- We use a set of `Transition`s to represent the set of transitions.

We follow the algorithm described in Section 4.5 to maintain the probabilistic transition system.

- As we described in Proposition 4.5.1, the probabilistic transition system contains an initial state s_0 and a sink state $s_\perp$, We introduce a state `INITIAL` with value 0 and a state SINK with value -1.

- In the event `stateAdvanced` (see Section 7.1) when the new state is not final, we add a transition to the sink state as described in Theorem 4.5.2.

- We use an object named `sinkTransitions` of type `HashMap` to store all transitions leading to the sink state. In this map, the key is the id of the source state of a transition. Thus we can quickly locate such a transition.

Now that we have discussed how the probabilistic transition systems of Proposition 4.5.1 and Theorem 4.5.2 are represented, we shall show how we can use those probabilistic transition systems to compute the progress measure in the next section.

7.4 The Computation of the Progress Measure

In order to compute the progress measure, we use the model checking tool MRMC. MRMC is a model checker developed by the Software Modeling and Verification

(MOVES) group at RWTH Aachen [17]. It is capable of model checking probabilistic transition systems (also known as discrete time Markov chains). MRMC takes as input a file, which contains transitions. Each transition consists of the source and target states and its probability. The states are represented by natural numbers and the probability is represented by a real number in the interval $[0, 1]$. MRMC can compute the probability of reaching a particular state.

As we have seen in Section 4.3, the progress measure for invariants can be characterized in terms of probability of reaching the state $s_\perp$ of the searched system. Hence, we can use MRMC to compute the progress measure for invariants.

On the one hand, MRMC is written in C. On the other hand, JPF and our extension are developed in Java. To bridge the gap we use JNI. JNI is a two-way interface to combine Java code with code written in another language, also known as native code.

- It allows the native code to call a method in Java.
- It also allows Java code to call a function in native code.

By using JNI, the native code and the Java code are running in the same process. Thus we can reduce the "overhead of copying and transmitting data across different processes" [19]. And since we are using the same memory space, we can reduce the memory usage as well.

When invoking a function written in C through JNI, there are a number of conventions to follow. Consider, for example, the native method `double getMeasure(int, int)` in the Java class `PTS` of the package `probabilistic` which is to invoke MRMC to compute the progress measure. The corresponding function in C has the following signature

```
JNIEXPORT jdouble JNICALL Java_probabilistic_PTS_getMeasure(JNIEnv *,
jobject, jint, jint)
```

- The name of the C function starts with `Java_`, followed by its package name, its class name, and its method name. In this case, they are `probabilistic`, `PTS`, and `getMeasure`, respectively.
- The `JNIEXPORT` and `JNICALL` macros are part of the declaration of the function. `JNIEXPORT` is used to decorate the return type of the JNI function. And `JNICALL` is used to decorate the JNI function. For more details, we refer the reader to [19, Section 12.4]
- The types in the Java signature are mapped to types in C. For example, `int` is mapped to `jint`, `double` is mapped to `jdouble`, `boolean` is mapped to `jboolean`, `object` is mapped to `jobject`, and so on. For more details, we refer the interested reader to [19, Chapter 12].

- The function in C has two additional parameters. These are the first two and have types of `JNIEnv*` and `jobject`. The `JNIEnv` pointer refers to a set of JNI functions, which can be used to access the JVM. And the `jobject` refers to the object which calls the C function. In our example, this `jobject` is a `PTS` object.

We develop a set of functions within the JNI framework to use MRMC to compute the progress measure.

- MRMC uses a sparse matrix to store the transitions. Instead of constructing this matrix entirely each time we compute the progress measure, we keep the matrix in memory and incrementally add transitions to the matrix.

- The capacity of a matrix is determined by the maximum numbers of rows and columns it can hold. For efficiency reasons, we double the number of rows and columns, whenever the matrix reaches its capacity.

- We set all necessary arguments so that MRMC can run without user interference.

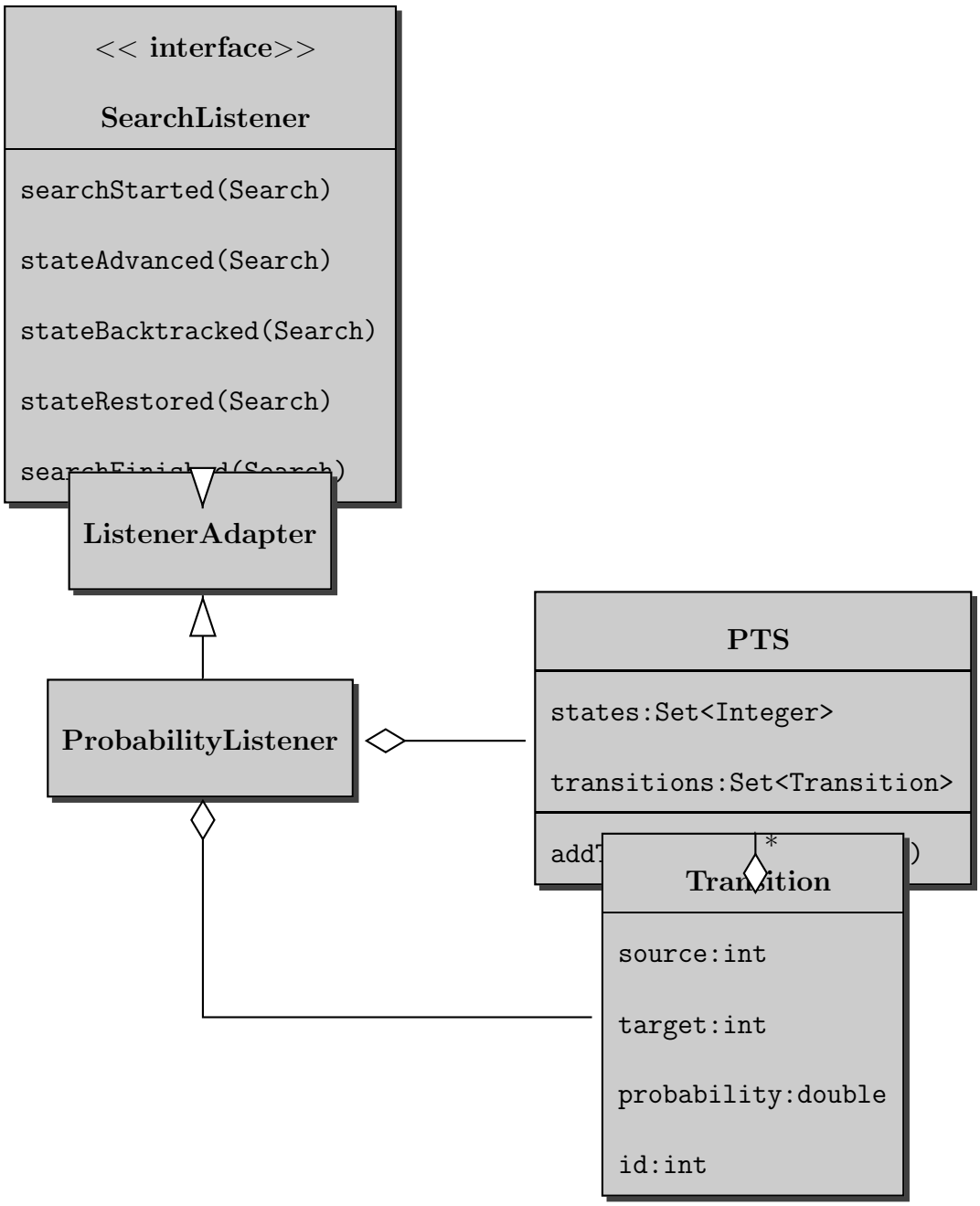
- We load the library `mrmcCJavaInterface`, which holds the above JNI methods used to access MRMC, in our `PTS` constructor instead of loading it in a static context. Thus we are able to have a single instance of the library for each `PTS` object.

- We declare this set of JNI methods in the class `PTS` to compute the progress measure.

- We free the memory allocated for the matrix at the end of model checking.

7.5 The Complete Picture

In the previous sections we have discussed all the classes needed to compute the progress measure for invariants. Here we put them together in the following UML diagram to show their relationships.



When we use JPF to model check the class `BiasedDie` (see Example 5.1.1),

which also includes a counter to record the number of times the coin is flipped. We set up the configuration file as follows, which uses breadth-first search and registers our search listener `ProbabilityListener`.

```
listener = probabilistic.ProbabilityListener

search.class = gov.nasa.jpf.search.heuristic.BFSHeuristic

probability.maximum.transitions=100

probability.measure.sampling_interval=5
```

If we run JPF with its progress measure enabled, it will report the progress for every 5 transitions, up to a maximum of 100 transitions. Below, we show the output produced by JPF.

```
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center



============================== system under test

application: randomized/BiasedDie.java


============================== search started: 5/4/10 4:20 PM

free=6055560,  total=11010048,  max=2524381184
```

0: 0.0

5: 0.0

10: 0.273

15: 0.6300000000000001

20: 0.6356700000000001

25: 0.8295000000000001

30: 0.8295000000000001

35: 0.8468376000000001

40: 0.917427

45: 0.917518854

50: 0.95962671

55: 0.95962671

60: 0.97175070363

65: 0.9802249611

70: 0.98022673257

75: 0.990310939527

80: 0.99031098735669

85: 0.99525242414115

90: 0.99525242414115

95: 0.9956797177345502

```
100: 0.9976736935687263


============================== results

no errors detected


============================== statistics

elapsed time:       0:00:00

states:             new=54, visited=47, backtracked=101, end=48

search:             maxDepth=19, constraints=0

choice generators:  thread=1, data=50

heap:               gc=101, new=271, free=530

instructions:       3749

max memory:         17MB

loaded code:        classes=72, methods=857


============================== search finished: 5/4/10 4:20 PM
```

8 Implementation of the Search Strategies

In Chapter 5 we discussed our three new search strategies: priority-first search (PFS), breadth-first priority-second search (BFPSS) and randomized search (RS). All three of these search strategies take the probabilities of the enabled transitions into account to choose the next transition. In this chapter we shall discuss our implementation of these three search strategies in JPF.

8.1 The Abstract Classes Search and ProbabilitySearch

In JPF, a search strategy is implemented as a class. This class has to extend the abstract class `Search`. In the class `Search`, there are multiple attributes and methods, which can be accessed directly from a subclass. Next, we discuss those that we shall use later in this chapter.

- The attribute `vm` of type `JVM` refers to JPF's JVM. The `JVM` class has methods such as `getState` to extract the `VMState` which JPF currently checks, and `getPathLength` to extract the length of the execution path along which the

current state has been found by JPF.

- The boolean attribute `done` indicates whether there are no unchecked transition left. It can be used by listeners to terminate the search process.

- The method `hasPropertyTermination` returns true when a property has been violated and the attribute `done` is true as well. Otherwise it will return false to continue the search.

- The integer attribute `depth` keeps tracks of the length of the path along which the current state has been found. This corresponds to the level as used in search strategies such as BFS and BFPSS discussed in Chapter 5.

- The boolean attributes `isNewState`, `isEndState`, and `isIgnoredState` specify whether the current `VMState` is new, is final and should be ignored, respectively. A state is ignored when JPF encountered a violation of a property in the state and JPF is configured to detect multiple violations.

- The method `forward` invokes the `JVM` method `forward` and returns a `boolean` to indicate whether there are any more enabled transitions.

- The abstract method `search` is used by JPF to drive the model checking process as we shall describe. A concrete search class overrides this method to specify the order in which transitions are explored.

A search strategy can be registered through the configuration file `jpf.properties`. For example, to use DFS, we add to the file `jpf.properties` the following line.

```
search.class = gov.nasa.jpf.search.DFS
```

During run time, JPF uses a concrete search class rather than the abstract class `Search`. In a concrete search class, the method `search` usually contains a loop, through which JPF can systematically explore the state space by using the three methods of the `JVM` class: `forward`, `backward`, and `restoreState`.

- A search strategy can use `restoreState(`s`)` to instruct JPF to jump to the state s.
- A search strategy can use the method `forward` to instruct JPF to advance to another state. When there is an unvisited transition enabled in the current state, JPF executes the sequence of bytecode instructions that takes it to the other state and returns true; otherwise JPF simply returns false to indicate there are no more transitions enabled.
- A search strategy can use the method `backtrack` to instruct JPF to go to the previously visited state.

Furthermore, since all our new search strategies take the probabilities of transitions into account, we developed our own abstract search class `ProbabilitySearch`,

which is an extension of the class `Search`. This abstract class contains the common part of our new search strategies. For example, we introduce an attribute `collection` of the type `Collection`. This collection contains `ProbabilisticTransitions`. A `ProbabilisticTransition` contains the source state, the target state, the depth of the target state, and the probability of the execution path from the initial state to the target state of this transition. We do not instantiate the attribute `collection`. Instead a concrete class which extends this class should instantiate the attribute and specify its type, for example, a priority queue or a set. As in the implementation of DFS and BFS in JPF, we restrict the depth of the search. For that purpose, JPF introduces a property in the configuration file. The value of this property can be retrieved by means of the method `getMaxSearchDepth` of the `Search` class. Moreover, we introduce other attributes such as `current` of type `VMState` to keep track of the current state, and `transition` of type `ProbabilisticTransition` to store and retrieve the information of the selected transition.

In the class `ProbabilitySearch` and its extensions we follow the coding convention of JPF. We notify a registered listener when we trigger an event such as `stateAdvanced` and `stateRestored`. Also we keep checking the system properties of JPF, such as the attribute `done` and the method `hasPropertyTermination` so that a search strategy can react to other JPF components. Recall that we use events to implement the computation of progress as discussed in Chapter 7.

An extended concrete class can override methods to determine the order in which the transitions are explored. Our abstract class `ProbabilitySearch` implements the abstract method `search`. Its code is shown below.

```
1  public void search () {
2      this.current = super.vm.getState();
3      super.notifyStateStored();
4
5      super.done = false;
6      super.notifySearchStarted();
7
8      this.transition = null;
9
10     if (!super.hasPropertyTermination()) {
11       this.generateTransitions();
12       while (!this.collection.isEmpty() && !super.done) {
13         this.selectTransition();
14         this.generateTransitions();
15       }
16     }
17     super.notifySearchFinished();
```

```
18 }
```

This `search` method contains a loop (line 12 – 15), in which we extract a transition from `collection` by using `selectTransition` and assign it to `transition`, and create new transitions by using `generateTransitions` and add them to `collection`. Here we use the following methods.

- The method `expandState` extracts the target state from the selected `transition` and invokes the `JVM` method `restoreState` to instruct JPF to jump to the target state. The code is listed below.

```
protected void expandState() {
    this.current = this.transition.getTarget();
    super.vm.restoreState(this.current);
    super.depth = super.vm.getPathLength();
    super.notifyStateRestored();
}
```

- The method `generateTransitions` extracts all transitions enabled from the current state. We invoke the method `forward` to visit another state. If the target state is a new state and satisfies some other conditions, such as not being a final state and not exceeding the maximal depth specified in the configuration file, we shall create a new `ProbabilisticTransition` and add it to the `collection`. Its code is listed below.

```
protected void generateTransitions () {
    while (!super.done) {
      if (!super.forward()) {
        super.notifyStateProcessed();
        return;
      }

```

```
8         super.depth++;
9         super.notifyStateAdvanced();
10        if (super.hasPropertyTermination())
11          return;
12
13        if (!super.isEndState && !super.isIgnoredState) {
14          if (super.isNewState && super.depth >= super.
                getMaxSearchDepth()) {
15            super.notifySearchConstraintHit(DEPTH_CONSTRAINT);
16          } else {
17            double probability = this.computeTransitionProbability
                  ();
18            ProbabilisticTransition t = new
                  ProbabilisticTransition(this.current, super.vm.
                  getState(), super.depth, probability);
19            this.collection.add(t);
20          }
21          super.notifyStateStored();
22        }
23        this.backtrackToParent();
```

```
24        }
25    }
```

- We define an abstract method called `selectTransition`; it is used to retrieve and remove a transition t from the `collection`. Since each search strategy uses a different data structure, we leave it to a concrete search class to implement this method.

- The method `backtrackToParent` invokes the `JVM` method `backtrack` to backtrack to the previously visited state.

In the method `generateTransitions` we use the method `computeTransitionProbability` to compute the probability of a transition. In Chapter 6 we showed how to extend JPF to a probabilistic model checker. In such a probabilistic model checker, each transition has a probability associated with it. We invoke the method `getProbability` of the interface `Probable` to retrieve the probability of a transition t enabled in the current state. This probability is multiplied by the probability of the execution path from the initial state to the current state (the latter probability is obtained in line 9). This product captures the probability of the execution path from the initial state to the target state of the transition t. The code is shown below.

```
1  protected double computeTransitionProbability(){
```

```
2       double probability = 1.0d;
3       ChoiceGenerator cg = super.vm.getChoiceGenerator();
4       if (cg instanceof Probable) {
5           probability = ((Probable)cg).getProbability();
6       }
7
8       if (this.transition != null){
9           probability=probability * this.transition.getProbability();
10      }
11      return probability;
12  }
```

8.2 The Classes PFS and BFPSS

Recall that probability-first search (PFS) sorts the enabled transitions by their probabilities, and breadth-first probability-second search (BFPSS) is an enhancement of BFS in which transitions at the same level are sorted by their probability. Our concrete search strategies `PFS` and `BFPSS` extend the class `ProbabilitySearch`. The abstract class `ProbabilitySearch` does not have a concrete data structure for the attribute `collection`, which is left for a concrete search class to implement.

We have discussed in Chapter 5 how we use a priority queue to store the collection of transitions in these two search strategies. We create a priority queue with a `Comparator`. This `Comparator` provides the definition of the ordering among transitions. The `Comparator` class contains the `compare` method. Recall that `compare`(t_1, t_2) has to return zero if t_1 and t_2 are equal (according to the ordering defined in Section 5.2 and 5.3); a negative number if t_1 is smaller than t_2 (with respect to the ordering); and a positive number if t_1 is greater than t_2. Since PFS considers only the probability when comparing transitions, in the `Comparator` we only compare the probabilities of the transitions. The code is listed below.

```
1  new Comparator<ProbabilisticTransition> () {
2      public int compare (ProbabilisticTransition t1,
           ProbabilisticTransition t2) {
3        double p1 = t1.getProbability();
4        double p2 = t2.getProbability();
5
6        if (Math.abs(p1-p2) < Double.MIN_VALUE)
7          return 0;
8        else if ( p1 > p2 )
9          return -1;
10       else
```

```
11          return 1;
12      }
13  }
```

BFPSS considers the depth of the transitions and their probabilities together, so in its `Comparator` we compare both. The code is listed below.

```
1   new Comparator<ProbabilisticTransition> () {
2       public int compare (ProbabilisticTransition t1,
            ProbabilisticTransition t2) {
3         int d1 = t1.getDepth();
4         int d2 = t2.getDepth();
5         double p1 = t2.getProbability();
6         double p2 = t2.getProbability();
7
8         if (d1 == d2 && Math.abs(p1 - p2) < Double.MIN_VALUE)
9           return 0;
10        else if (d1 < d2 || d1 == d2 && p1 > p2)
11          return -1;
12        else
13          return 1;
```

```
14          }
15        }
```

Furthermore, we have to implement the abstract method `selectTransition`. In this method, we remove a `ProbabilisticTransition` t from the priority queue using the method `poll`. We then use the method `expandState` to extract information of the transition and instruct JPF to jump to the target state. The code is listed below.

```
1     protected void selectTransition() {
2       this.transition = ((PriorityQueue<ProbabilisticTransition>)super.
            collection).poll();
3       super.expandState();
4     }
```

8.3 The Class RandomizedSearch

The third search strategy we proposed in Chapter 5 is randomized search (RS). RS randomly selects an enabled transition. The chance that such a transition is selected is proportional to its probability. Our concrete search class `RandomizedSearch` extends the abstract class `ProbabilitySearch` as well. In Chapter 5 we showed that the worst case running time of RS is $O(|T_0| \log(|T_0|))$ if we represent the set

as an augmented red-black tree. However, `RandomizedSearch` simply uses a set of type `LinkedList` as `Collection` to store the enabled transitions.

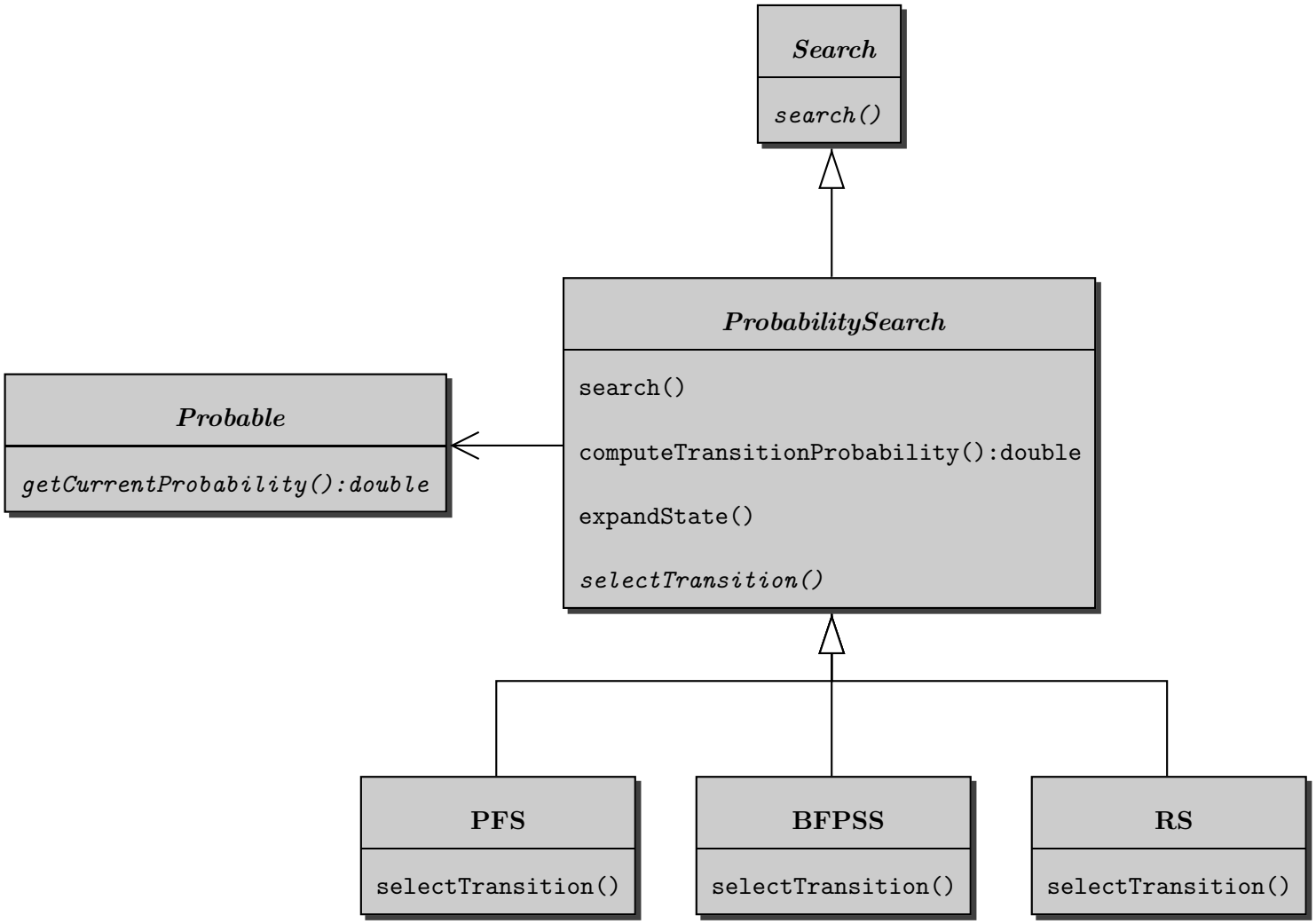
We have implemented the method `selectTransition` in class `RandomizedSearch`. First we sum up the probabilities of the transitions in the collection to `totalProbability`. Next we generate a random `double` in the interval $[0, \texttt{totalProbability}]$. This allows us to randomly choose a transition t from the collection with the probability $\frac{\text{prob}(t)}{\texttt{totalProbability}}$. For example, if there are two transitions in the collection, such that $\text{prob}(t_1) = 0.3$, and $\text{prob}(t_2) = 0.5$, then the `totalProbability` is 0.8. We choose t_1 with probability 0.375, and choose t_2 with probability 0.625. We compute `totalProbability` each time when we make choice. Thus the running time is $O(|T_0|^2)$. The code of `selectTransition` is listed below.

```
1  protected void selectTransition () {
2    double totalProbability = 0.0;
3    for (ProbabilisticTransition t : super.collection){
4        totalProbability += t.getProbability();
5    }
6    double choice = random.nextDouble() * totalProbability;
7
8    int i  = 0;
```

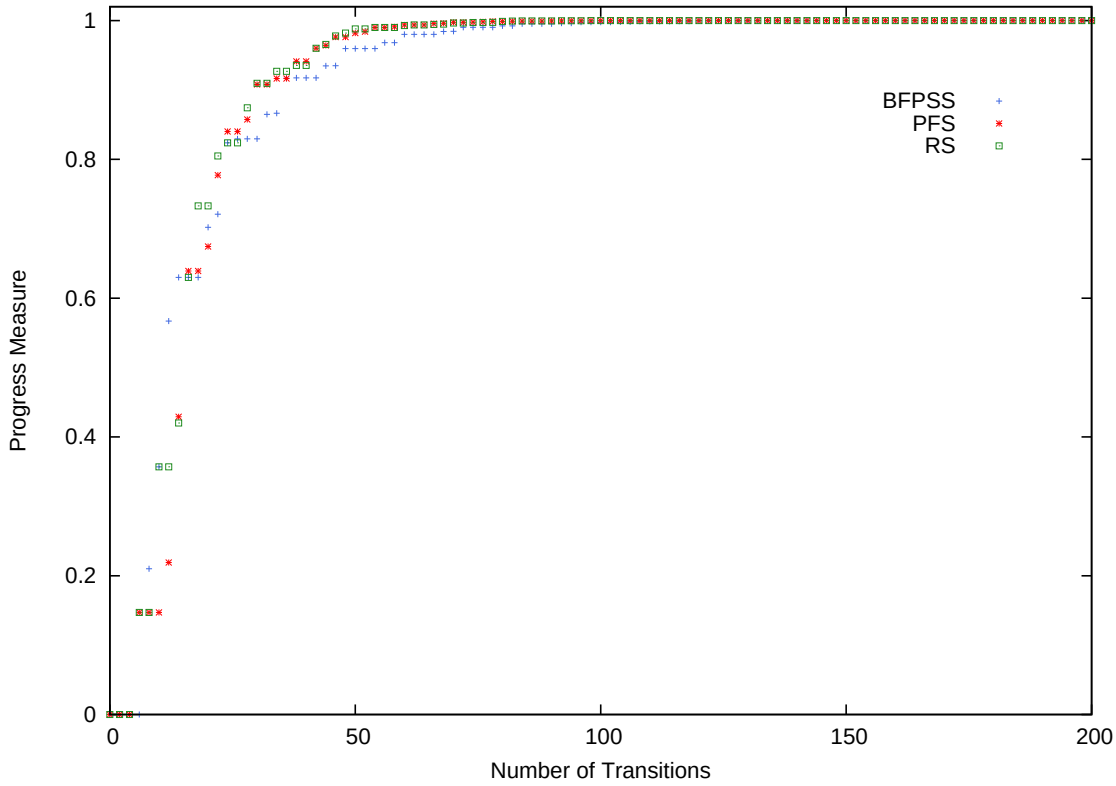
```
9       double sum = 0.0;
10      for (ProbabilisticTransition t : super.collection){
11          sum += t.getProbability();
12          if (sum >= choice || i == super.collection.size() - 1){
13              super.transition = t;
14              ((ArrayList)super.collection).remove(i);
15              break;
16          }
17          i++;
18      }
19
20      super.expandState();
21    }
22  }
```

8.4 The Complete Picture

In the previous sections we have discussed all the classes needed for JPF to implement the search strategies introduced in Chapter 5. The UML diagram below shows the relationships among them.



Next, we use the three search strategies to model check the Java bytecode of the `BiasedDie` class (see Section 5.1), which also includes a counter to record the number of times the coin is flipped. The following diagram shows the resulting progress for 200 transitions. Again this result confirms that the search strategies are not comparable.



So far, we have discussed the implementation of our theory. In the next chapter, we shall show how to use our extension of JPF to keep track of its progress when model checking randomized Java code.

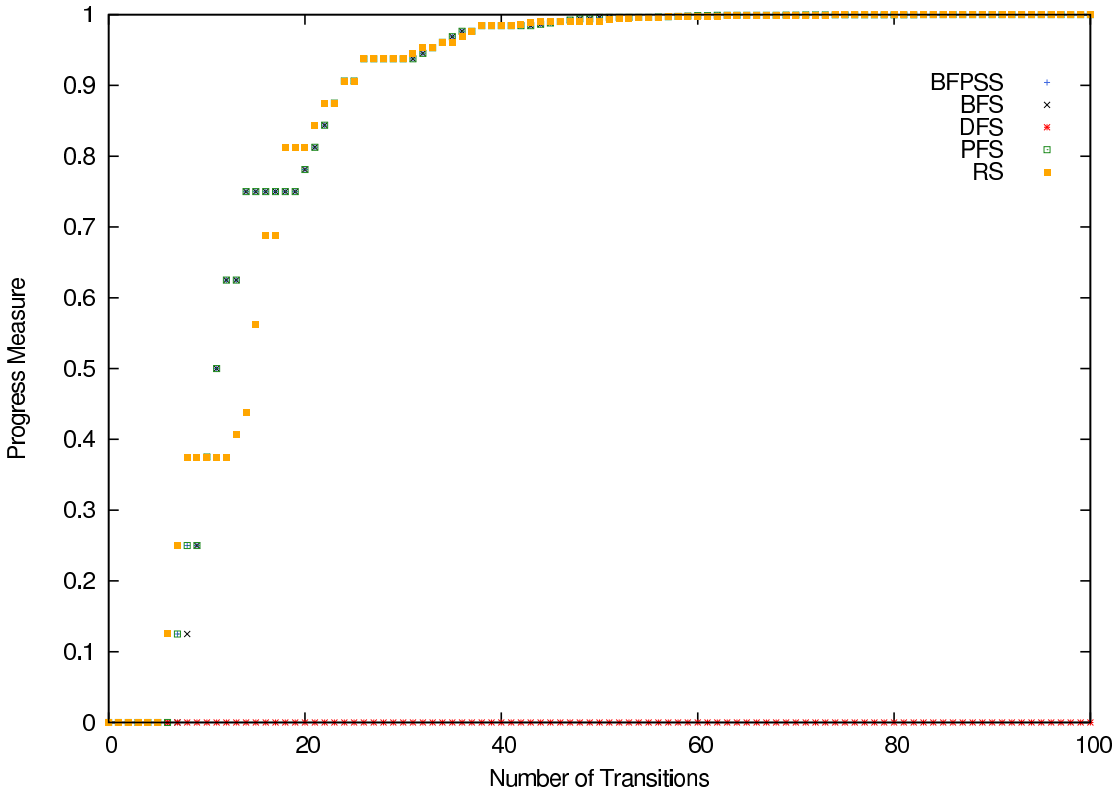
9 Case Studies

In the previous chapter we presented the implementations of our three new search strategies. As we already discussed in Chapter 5 the amount of progress made by the search strategies is incomparable. In this chapter we shall compare the amount of progress different search strategies make when model checking implementations of seven randomized algorithms.

When we use JPF to model check the implementation of randomized algorithms, we enable our probabilistic extension of JPF to compute the progress. We run JPF for all five search strategies: JPF's DFS and BFS, and our three new ones PFS, BFPSS and RS. For each algorithm we plot the amount of progress against the number of transitions checked by JPF for each of the search strategies. For these graphs, all search strategies, including RS, are run once only. We also show the time it takes JPF to make $0.1, 0.2, \ldots, 0.9, 0.99, 0.999, \ldots$ progress. We run JPF five times for each search strategy and report the average time. If a search takes more than an hour, we stop it.

9.1 Die and Biased Die

In [18], Knuth and Yao showed how a fair die can be implemented by means of a fair coin. We implemented this randomized algorithm in Java and added a variable to record the number of times the coin was flipped. We then used JPF to model check the Java bytecode for uncaught exceptions. The diagram below shows the relationship between our progress measure and the number of transitions. Note that we only show the first 100 transitions because after that the progress is so close to 1.0 that we cannot see any difference in the diagram.



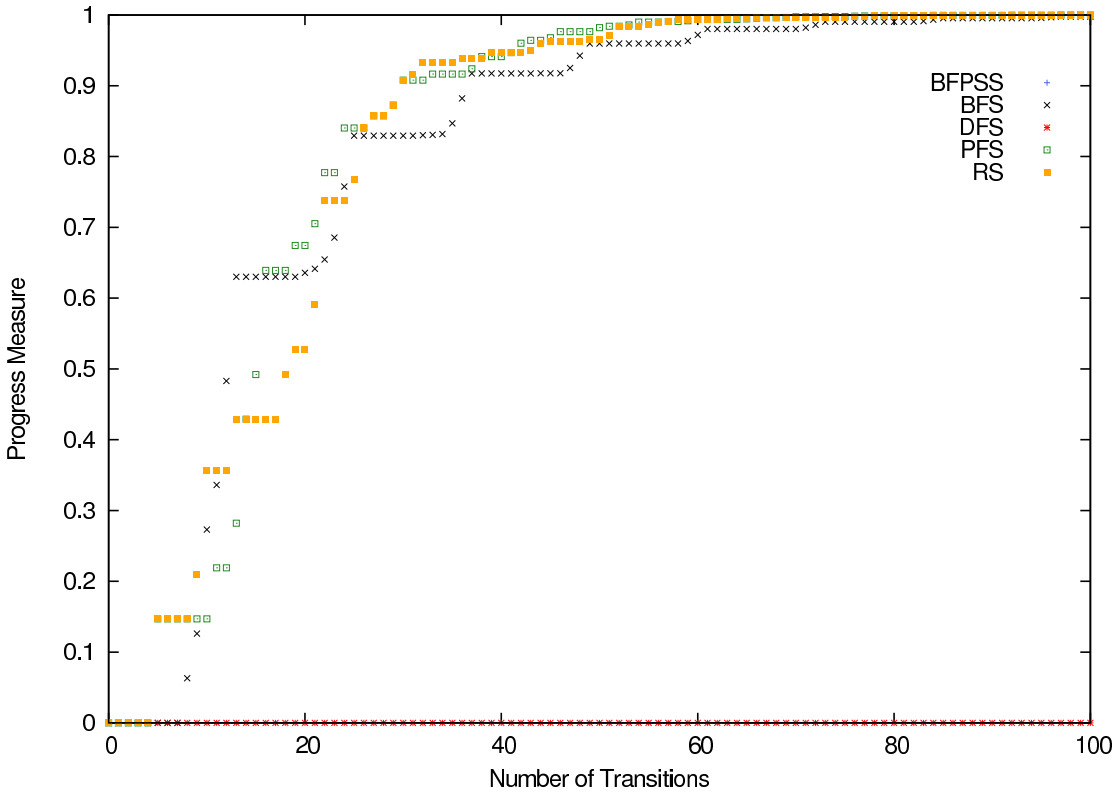
From the diagram, we can see that DFS makes no progress at all. This algorithm gives rise to extremely long execution paths, in which the number of transitions is bounded by the number of values of type `long` in Java. DFS searches one of these execution paths. On the one hand, BFS, PFS and BFPSS are almost identical because the coin is fair, thus all outcomes of the die have the same probability. As a consequence all execution paths have the same structure in terms of their probability. This means that all three of these search strategies make roughly the same amount of progress. The following chart shows the running time (in milliseconds) to achieve different values of the progress measure.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.99	$1-10^{-9}$
DFS											
BFPSS	155	192	224	231	237	244	248	251	259	263	305
PFS	158	189	214	229	238	245	253	257	262	270	303
RS	168	194	215	241	245	250	256	262	269	273	313
BFS	164	195	219	231	241	249	255	262	266	274	311

Since DFS makes no progress, we cannot report the time. The other four search strategies take almost the same amount of time to make a certain amount of progress because the die is fair. However we notice that none of the search strategies can finish model checking because of the extremely long execution paths we discussed

above. After roughly one hour of running, we stop all searches.

Furthermore when we replace the fair coin with a biased coin, which has probability 0.7 of heads and 0.3 of tails, we obtain a biased version of a die. When we model check this algorithm, comparing it with the regular die, the search strategies make very different amounts of progress as we show in the following diagrams. Again we only show the first 100 transitions in the diagram below.



As we can see, DFS still does not make any progress for the same reason as for the regular die. Other than DFS, in the first 10 transitions, PFS makes relatively slow progress in terms of our measure because the transition of one long execution

path has a high probability, which causes our PFS to prefer this long execution path at first. Then PFS starts to choose other execution paths, since the probability of the long execution path decreases. Moreover BFPSS is faster than BFS because BFPSS chooses the high probability transitions at the same level first. The running time (in milliseconds) is shown in the following chart.

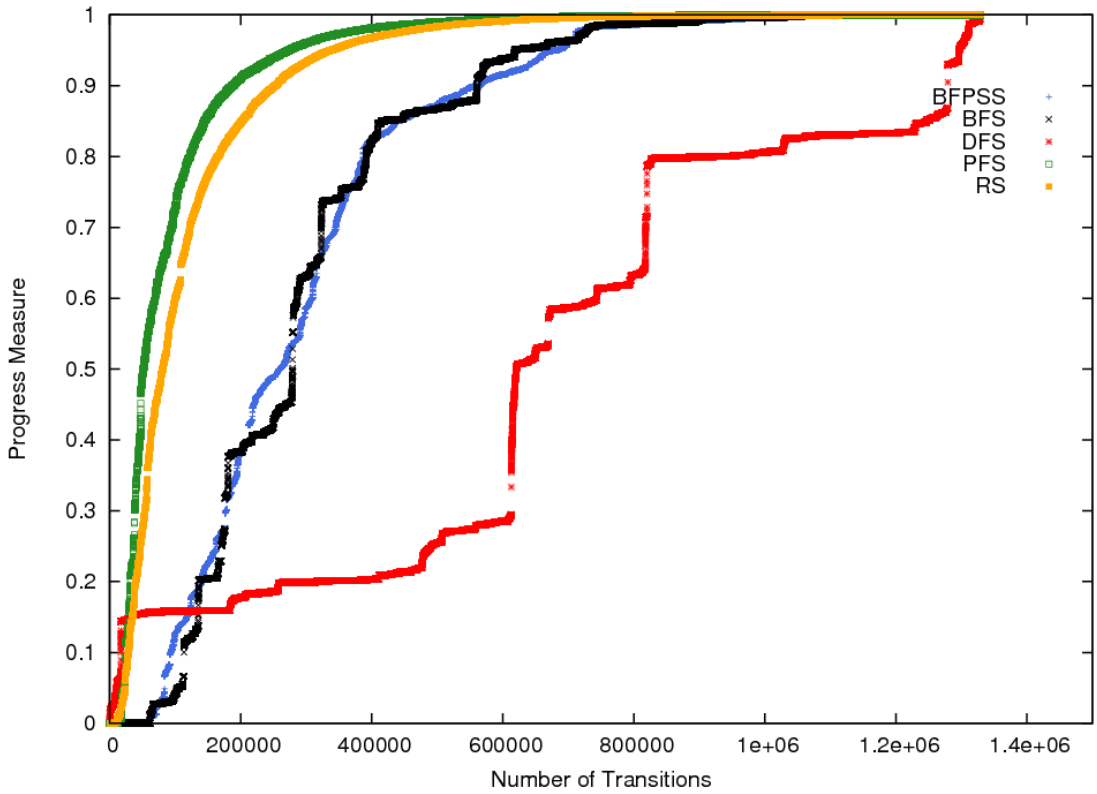
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.99	$1-10^{-9}$
DFS											
BFPSS	177	206	246	257	264	276	279	284	288	294	397
PFS	163	193	216	234	245	249	256	259	265	275	338
RS	169	209	230	253	261	268	273	277	280	290	342
BFS	164	187	204	227	234	247	252	259	266	274	371

From the above chart we can see that BFPSS uses more time than BFS because BFPSS needs more time to sort the transitions so that it can choose the next transition as we discussed in Chapter 5. PFS takes less time to make progress because it can make more progress by searching fewer transitions.

9.2 Randomized Quicksort

The second algorithm which we implemented and model checked is a randomized version of quicksort introduced by Hoare [14]. In this algorithm in order to sort an array of objects, we select the first element as pivot, and divide the array into two: an array in which the objects are smaller than the pivot; and an array in which the objects are greater than the pivot. Then we recursively sort these two arrays unless there is either no or only one object in these arrays in which cases the array is sorted. After we put all the objects back into one array, we get a sorted array. In the worst case, when an array of size n is already sorted, quicksort makes $O(n^2)$ comparisons. The randomized version chooses a random pivot instead of a fixed one. This lowers the expected number of comparisons needed to sort the array. In our implementation we try to sort an array of 14 integers. When we model checked our implementation, all search strategies ran out of memory after roughly 1,300,000 transitions.

As we show in the diagram below, PFS makes the fastest progress because PFS always chooses the execution path with highest probability. BFPSS is better than BFS because BFPSS chooses higher probability transitions at the same level. DFS makes the slowest progress because in this algorithm, the first several execution paths which DFS searches have lower probability.

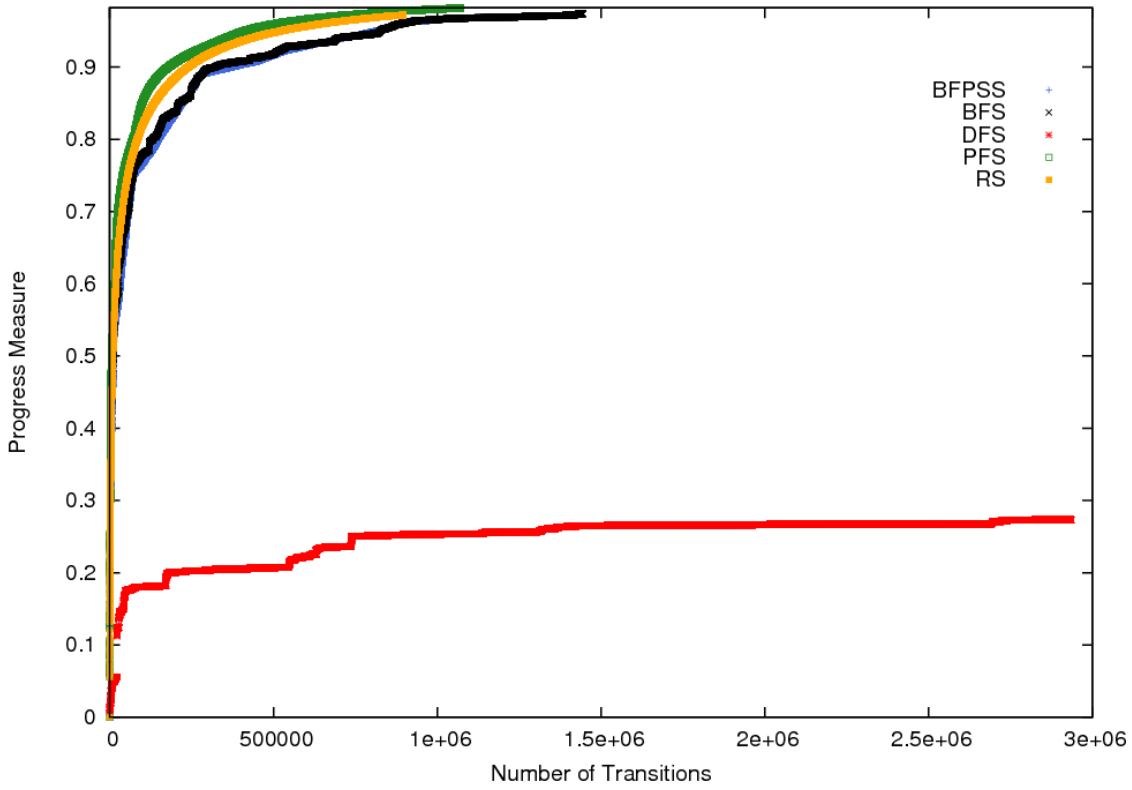


The table below of the running time (in seconds) shows that PFS makes progress the fastest. It takes BFPSS and BFS longer time to make the same progress because these two search strategies explore the transitions level by level. Since DFS always tries to explore one execution path before another and has no knowledge of probabilities, at first it takes more time for DFS to make progress than BFPSS and BFS. However it catches up with BFS and BFPSS after it searches some execution paths with high probability. Since the system is too large for any search strategy to finish its search within an hour, we stopped them.

	0.1	0.2	0.3	0.4	0.5	0.8	0.9	0.99	0.9999	0.99999
DFS	21	408	868	870	880	1400	2136	2228		
BFPSS	107	185	263	352	498	977	1865	3708		
PFS	26	33	39	47	54	152	266	1147	3476	3934
RS	40	60	87	120	166	483	864	3145		
BFS	141	179	253	354	527	905	1702	3489		

9.3 Random Select

The next randomized algorithm is randomized select. This algorithm is to find the n^{th} smallest object from an array. Like randomized quicksort, we randomly choose a pivot, and divide the array into two by comparing with the pivot. Assume that the size of the array with the smaller objects is k. Then either the pivot is the n^{th} object if $k = n - 1$; or recursively we find the n^{th} object in the array of smaller objects if $k > n$; or the $(n - k - 1)^{th}$ object in the array with larger objects. In the implementation we try to find the 6^{th} of 20 integers. The following diagram shows the progress measure of model checking.



As expected, the results are very similar to random quicksort. PFS makes the fastest progress, followed by BFPSS and BFS. DFS makes the slowest progress in terms of the progress measure. RS makes progress bounded by PFS and BFS. On the one hand DFS runs out of memory after it searches 3,000,000 transitions, and can only make progress of less than 0.3. On the other hand, PFS, BFS, BFPSS and RS run out of memory after searching 1,000,000 to 1,500,000 transitions, but all of them make progress between 0.98 and 0.99.

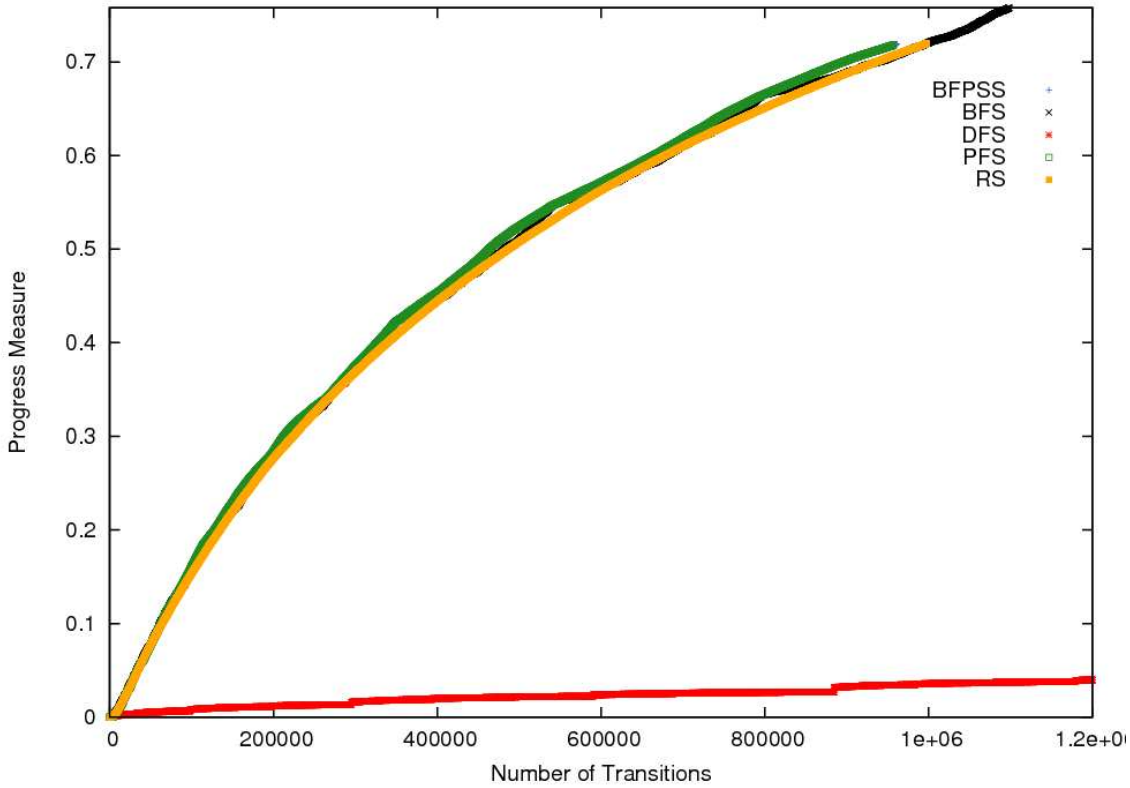
The running time (in milliseconds) chart below shows a similar pattern. After an hour, DFS only makes progress of 0.2, and BFS, PFS, BFPSS and RS all

make progress of 0.9. Among these four search strategies we can see that PFS is the fastest to make progress, and RS uses more time due to the time needed to compute the sum of the probabilities as we discussed in Section 8.3. BFPSS makes slower progress than BFS because the priority queue used by BFPSS needs time to keep the enabled transitions sorted.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
DFS	12344	90719							
BFPSS	965	1555	2617	5769	8656	20786	38746	153601	800357
PFS	914	1122	2288	3032	5007	10410	15075	46220	179341
RS	918	1610	2679	4623	7874	14048	30235	100350	661621
BFS	740	1608	2674	4980	9011	18022	35859	116643	562527

9.4 Skiplist

The next algorithm we shall present is a skiplist [24]. We do not present the details of the algorithm here, but we refer the interested reader to, for example [22, Section 8.3]. We randomly choose the height of each tower. Here we focus on adding 15 integers to an empty skiplist. The following diagram shows the progress measure of model checking the algorithm.



DFS makes the slowest progress among all five search strategies because at first it searches the execution paths with the lowest probability. After searching 1,200,000 transitions the progress it made does not exceed 0.1. PFS has a slightly better performance than BFPSS and BFS. Note that all our search strategies ran out of memory after searching 950,000 to 1,200,000 transitions.

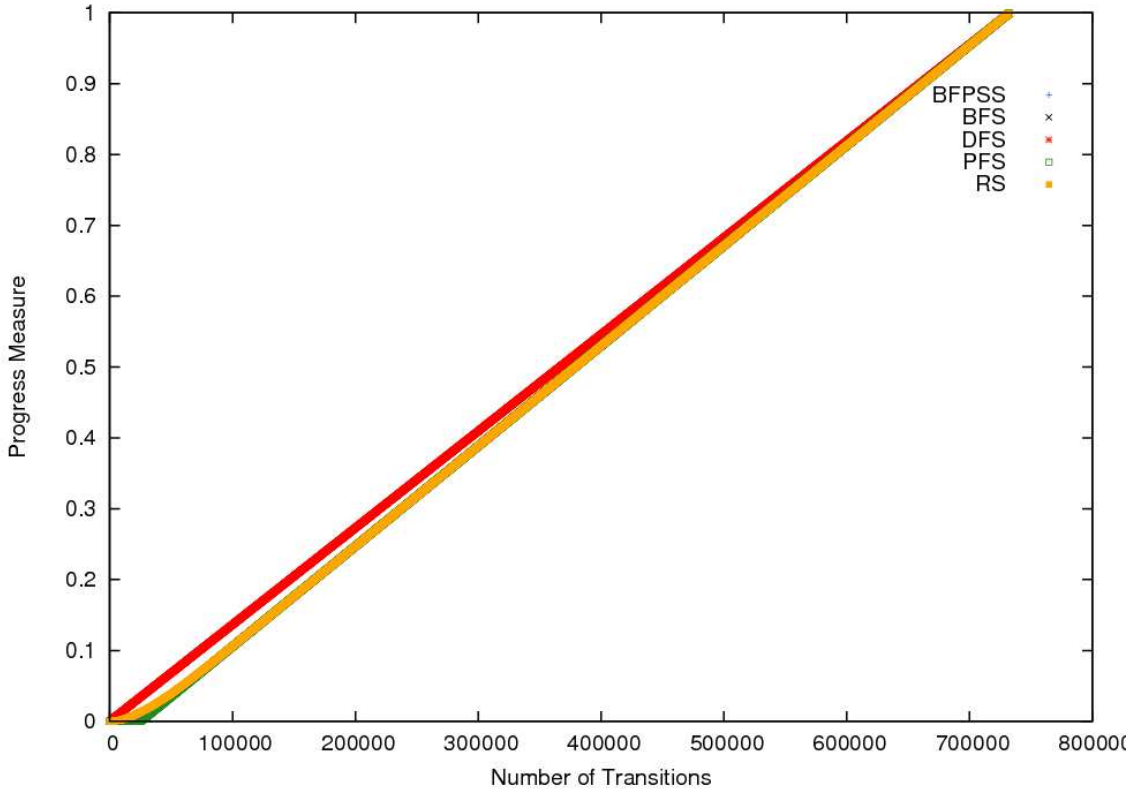
In the running time chart (in seconds) below, we see that DFS does not reach 0.1 within one hour for the same reason as we discussed in the previous section. RS takes more time to make a random choice as we explained in Section 8.3 and for this reason RS makes progress between 0.4 and 0.5 in an hour. Both PFS and

BFS make progress of 0.6 and PFS is slightly better than BFS. BFPSS is slower than BFS and reports a progress of 0.5 after an hour.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
DFS									
BFPSS	57	180	465	1054	2105				
PFS	43	132	364	853	1727	3442			
RS	83	343	928	2062					
BFS	47	137	350	828	1777	3459			

9.5 Random Matrix Equation

The next randomized algorithm which we introduce is random matrix equation [22, Theorem 7.1]. In this algorithm, assume that there are three $n \times n$ matrices A, B, C, and in order to verify whether $AB = C$, we randomly create a vector r of size n and test whether $ABr = Cr$. Assume that the values in the vector are randomly chosen in the range of $[1, m]$. If $ABr \neq Cr$ then obviously the algorithm reports $AB \neq C$. If, however the algorithm verifies that $ABr = Cr$, then the probability that $AB \neq C$ is at most $\frac{1}{m}$. In our implementation, we have $n = 4$ and $m = 29$. The progress of model checking is shown below.



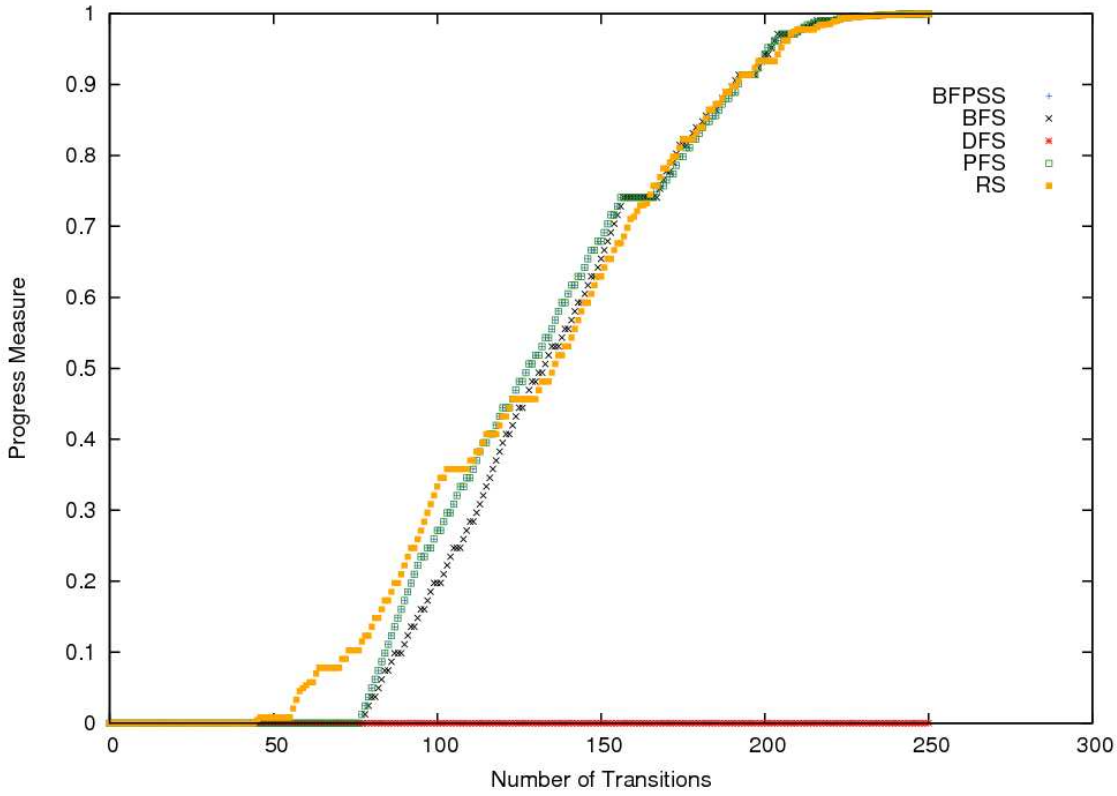
In this case all execution paths consist of a small number of transitions, and on the same level each transition has the same probability. Thus for all five search strategies there is an almost linear relationship between the progress being made and the number of transitions being checked. DFS is slightly better than the other four search strategies because it always explores one execution path before moving to another. On the other hand, there is no visible difference among PFS, BFPSS and BFS. The running time (in seconds) is listed in the following chart. All five search strategies have similar running time. However, BFS takes the least amount

of time to make progress, followed by PFS. Both of them reported a progress of 0.9 after an hour. And RS is the slowest of all five search strategies.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
DFS	492	943	1369	1795	2203	2613	3025	3456	
BFPSS	588	1144	1699	2301	2884	3453	3948		
PFS	437	859	1267	1674	2081	2489	2898	3307	3620
RS	507	993	1533	2034	2485	2956	3394	3837	
BFS	432	842	1255	1671	2080	2490	2897	3304	3622

9.6 Scissors Game

The next randomized algorithm which we shall model check is called the scissors game [22, Section 2.2.1]. In this algorithm, we simulate two players playing the scissors-paper-rock game. Players randomly choose scissors, paper, or rock. The rule to determine the winner is that rock beats scissors, scissors beats paper, and paper beats rock. We do not only determine the winner, but also record the number of wins for both players. The following diagram shows the progress of model checking.



Since there are extremely long execution paths in which the number of transitions is bounded by the number of values of type `long` in Java and DFS searches one of these paths first, it cannot make any progress at all. This also means that PFS makes slower progress than BFS and BFPSS because PFS will search a prefix of these execution paths as well. BFS and BFPSS make similar progress. After 250 transitions, BFS, BFPSS, PFS and RS all make progress very close to 1.0, so we only show the first 250 transitions in the diagram. The running time (in milliseconds) shows the same result.

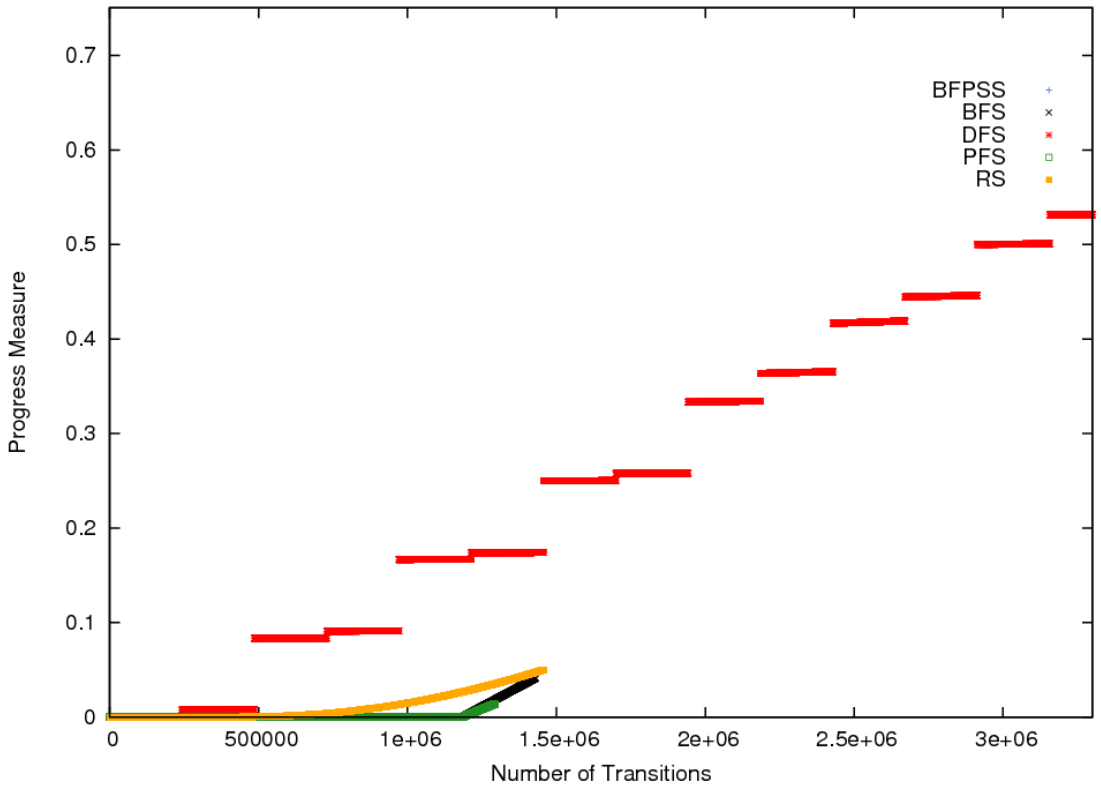
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.99	$1 - 10^{-9}$
DFS											
BFPSS	283	286	291	293	296	300	317	342	368	424	1107
PFS	406	409	411	414	417	429	460	489	628	709	2061
RS	299	301	304	307	310	365	407	454	548	688	2037
BFS	278	282	285	289	295	299	313	329	357	418	1066

On the one hand BFS runs the fastest since it does not need to sort the transitions. On the other hand PFS makes slow progress, and hence it uses more time to model check. Although BFS and BFPSS make the same amount of progress per transition, BFPSS takes more time to make progress since it needs time to sort the transitions in its priority queue.

9.7 Stable Marriage

The next algorithm we shall present checks how to use the proposal algorithm [22, Section 3.5] to solve the stable marriage problem. In the stable marriage problem there is a set of men and women. Each of them has a preference list which contains people of the opposite sex in a decreasing order. The problem is to find each a partner so that the couple can maintain stable relationships. In other words, for any two couples $X \leftrightarrow x$ and $Y \leftrightarrow y$, it should not be the case that X prefers y

over x and y prefers X over Y. Otherwise X may break up with x, Y may break up with y, and X and y may form a new couple. The proposal algorithm lets every unmatched man propose to a woman in his preference list in order. A woman can either accept the proposal if she is not in a relation or her current partner is less preferable than the current proposer; or she can simply reject the proposal if she wants to maintain the current relation. We have four men and four women in the set. Furthermore we create a random preference list for each person. Then we use the proposal algorithm to match them one by one and check if they are in a stable relation. The progress of model checking is shown in the following diagram.



Since the probabilistic transition system corresponding to the Java bytecode is so huge, we are unable to fully model check the bytecode. DFS searches 3,000,000 transitions and then runs out of memory. All other search strategies run out of memory after approximate 1,500,000 transitions. DFS makes the most progress. Because in this system, there are no extremely long execution paths as in the scissor game, and all execution paths have the same structure in terms of probability, DFS can make faster progress.

In the following time chart (in seconds), apart from DFS, the other four search strategies do not make 0.1 progress. This is because of JPF ran out of memory

before it reaches a progress of 0.1.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
DFS	4476	10188	16483	26658	38674					
BFPSS										
PFS										
RS										
BFS										

9.8 Summary

As we have seen, whenever there is an extremely long execution path and DFS searches this path, it will stop making progress. However it uses a simple data structure to store the enabled transitions, and hence it needs less memory. On the other hand, BFS, PFS and BFPSS usually make progress. Since both PFS and BFPSS take the probabilities into account, they often can make faster progress when considering the number of transitions. For most cases, PFS is the fastest search strategy. Compared with BFS, BFPSS can make slightly faster progress when considering the number of transitions. However when the size of system grows, the difference becomes smaller. The overhead of sorting the transitions makes BFPSS use more time to model check. The curves of RS are always smoother than the

other four, and is usually close to the curve of PFS. Since RS makes a random choice among the enabled transitions, the probability that it searches an extremely long execution path is small.

In conclusion, to model check a randomized algorithm, we suggest using PFS first. If the model checker runs out of memory, we suggest trying DFS next.

10 Conclusion

10.1 Overview

In this thesis we introduced the notion of a progress measure and characterized the progress of a search made by such a probabilistic model checker when it verifies an invariant. This is the major contribution of this thesis. We have also shown an algorithm which we used to compute the progress measure for invariants. Furthermore, we introduced three new search strategies probability-first search, breadth-first probability-second search and randomized search, which all take the probabilities of transitions into account when choosing the next transition. We compared them with depth-first search and breadth-first search.

We have implemented our theoretical framework within JPF. We implemented a number of randomized algorithms in Java and used our extension of JPF to model check them and compute the progress measure. For most examples, our new search strategies made progress faster than JPF's original depth-first search and breadth-first search.

10.2 Future Work

In this thesis we took the first steps towards a theory to measure the progress of probabilistic model checkers and implemented this theory with a probabilistic extension of JPF. This work suggests numerous directions for future research. Next we briefly discuss some.

1. Currently the progress measure can only be applied to sequential randomized algorithms. One direction for future research is to extend our theory so that we can measure the progress when we model check concurrent randomized algorithms. In that case, we not only have to consider probabilistic choices, but we also have to deal with nondeterministic choices and, hence, we have to consider schedulers.

2. Our algorithm to compute the progress measure can only handle invariants. However there are many other linear time properties that we may want to verify. One may want to develop algorithms to compute our progress measure for other classes of linear time properties.

3. Although in most cases our new search strategies PFS and RS make faster progress than DFS and BFS, in other cases DFS outperforms other search strategies. One may want to develop other search strategies, possibly based on ideas from directed model checking [9].

4. We believe that our developed theory can be adapted to symbolic probabilistic model checkers. This is another direction for future research.

5. From our experiments, we found that it takes a lot of time to compute progress measure. As a consequence, one may want to exploit a GPU to compute progress. Also, we may consider distributing the computation over multiple CPUs.

6. We believe that we need to implement more randomized algorithms and use our extension of JPF to model check them and measure the progress by different search algorithms, so that we may categorize randomized algorithms and possibly determine the best search strategies for each category.

7. Although we model checked some algorithms with a large state space such as the stable marriage problem, one may want to model check algorithms with even bigger state spaces, and compare the progress for different search strategies.

8. In our case studies, we ran the model checker JPF only five times for each search strategy and used the average running time. However we found that there is relatively big variance among the running times. So in the future one may want to run them more often and statistically analyze the results.

Although the first steps towards measuring the progress of probabilistic model checkers have been made, a lot of work remains to be done in this new area of research.

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, 2008.

[2] P. Benyon-Davies. Information Systems Failure: The Case of the London Ambulance Service's Computer Aided Dispatch Project. *European Journal of Information Systems*, 4(1):171–184, August 1995.

[3] Patrick Billingsley. *Probability and Measure*. Wiley-Interscience, New York, 1995.

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, second edition, 2001.

[5] Richard de Neufville. The baggage system at Denver: Prospects and lessons. *Journal of Air Transport Management*, 1(4):229–236, December 1994.

[6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738 – 761, August 1994.

[7] E.W. Dijkstra. Structured programming. In J.N. Buxton and B. Randell, editors, *Software Engineering Techniques*, pages 84–88. NATO Science Committee, Rome, October 1969.

[8] Satadip Dutta. An introduction to Abbot. *Java Developer's Journal*, April 2003.

[9] Stefan Edelkamp, Viktor Schuppan, Dragan Bosnacki, Anton Wijs, Ansgar Fehnker, and Husain Aljazzar. Survey on directed model checking. In Doron A. Peled and Michael J. Wooldridge, editors, *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence*, volume 5348 of *Lecture Notes in Computer Science*, pages 65–89, Patras, Greece, July 2008. Springer-Verlag.

[10] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In Dragan Bošnački and Stefan Leue, editors, *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 230–239, Grenoble, France, April 2002. Springer-Verlag.

[11] Ted Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, Berkeley, 2006.

[12] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, first edition, September 2003.

[13] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.

[14] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 5(7):321, July 1961.

[15] Richard M. Karp. The probabilistic analysis of some combinatorial search algorithms. In J. F. Traub, editor, *Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity*, pages 1–19. Academic Press, Pittsburgh, 1976.

[16] Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1-3):165–201, November 1991.

[17] Joost-Pieter Katoen and Ivan S. Zapreev. Safe on-the-fly steady-state detection for time-bounded reachability. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, pages 301–310, Riverside, September 2006. IEEE Computer Society Press.

[18] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In J.F. Traub, editor, *Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity*, pages 375–428, Pittsburgh, 1976. Academic Press.

[19] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, first edition, 1999.

[20] J. L. Lions. Ariane 5 flight 501 failure report by the inquiry board. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, July 1996.

[21] Donald W. Loveland. *Automated theorem proving: a logical basis*. North-Holland, 1978.

[22] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, first edition, August 1995.

[23] Esteban Pavese, Víctor Braberman, and Sebastian Uchitel. My model checker died! How well did it do? In *Proceedings of 2010 ICSE workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, pages 33–40, Cape Town, May 2010. ACM.

[24] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communication of ACM*, 33(6):668–676, June 1990.

[25] M. O. Rabin. The complexity of nonuniform random number generation. In J.F. Traub, editor, *Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity*, pages 21–40. Academic Press, Pittsburgh, 1976.

[26] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, P. Grunbacher, editor. *Value-based software engineering*. Springer, first edition, 2006.

[27] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, June 1995.

[28] A. Sokolova, E.P. de Vink, and H. Woracek. Coalgebraic weak bisimulation for action-type systems. *Scientific Annals of Computer Science*, 19:93–144, 2009.

[29] Ana Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. PhD thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, November 2005.

[30] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[31] Adriaan Cornelis Zaanen. *Integration*. North-Holland, Amsterdam, The Netherlands, 1967.