2015

A formal language towards the unification of model checking and performance evaluation

Yaping Jing
Iowa State University

A formal language towards the unification of model checking and performance evaluation

by

Yaping Jing

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Andrew S. Miner, Major Professor
Gianfranco Ciardo
Samik Basu
Leslie Miller
Arka Ghosh

Iowa State University

Ames, Iowa

2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

First and foremost, I am deeply grateful to my PhD advisor Andrew Miner. Dr. Miner has been a steady influence throughout my Ph.D. career. He has been patient and supportive in times of new ideas and difficulties. His high scientific standards set an example. He has listened to my ideas during our weekly meetings, and discussions with him often led to key insights. He's also given me the freedom to work everywhere I feel comfortable and focused. I also appreciate he supported me to travel to conference abroad, and gave me meaningful feedbacks before my technical talk. Above all, I learned a lot from Dr. Miner; he is one of the most knowledgable professors I could get within the department.

I also have to thank the members of my PhD committee, Professors Gianfranco Ciardo, Samik Basu, Leslie Miller, and Arka Ghosh for taking time reading the long thesis. I will forever be thankful to Professor Leslie Miller. I am fortunate for having been a teaching assistant of Professor Miller for several courses. His truly kindness, trustworthy, and righteousness at the time of my trouble is invaluable.

I also would like to thank my fellow graduate students: Wen-Chieh Chang, Du-hong Cheng, for their helpful discussions during my graduate study; thank Benjamin Rittgers, Eric Tuns, Yujia Ge, Yi-li Wang, and Yaping Feng, for multiple moving help; I enjoyed the time when we were hanging out together.

Finally, this work is dedicated to my family with great love. My mom, sisters and brother never lost faith in me, always praying for me, and endured all difficulties in helping my dream come true.

ABSTRACT

In computer science, model checking refers to a computation process that, given a formal structure, checks whether the structure satisfies a logic formula which encodes certain properties. If the structure is a discrete state system and the interested properties depend only on which states to be reached, not on the time or probability to reach them, traditional temporal logics such as linear temporal logic (LTL) and computation tree logic (CTL) are powerful mathematical formalisms that can express properties such as "no collision shall occur in a traffic light control system", or "eventually, a service is completed". To express performance-dependability related properties over discrete state stochastic systems, these logics have evolved into quantitative model checking logics such as probabilistic linear temporal logic (PLTL), probabilistic computation tree logic (PCTL), and computation tree stochastic logic (CSL), etc., and can express properties such as "with probability at least 0.98, the system will not reach a deadlock state before time 100". While these logics and their model checking algorithms are powerful, they are inadequate in expressing complex performance measures, either because they are limited to producing only true/false responses (although in practice, a real valued response can sometimes be obtained for the outer-most path quantifier), or the computational complexity is too expensive to be practical.

To address these limitations, for this PhD work, we propose a novel mechanism with the following research aims: 1) Define general specification formalisms to express performance queries in real values while retaining the ability to express temporal properties. 2) Develop efficient mathematical algorithms for the proposed formalisms. 3) Implement the approach in tools and experiment on large-scaled Markov models for the analysis of example queries.

CHAPTER 1. INTRODUCTION

In the past, performance evaluation and reliability evaluation were two separate disciplines in computer science and engineering [9, 49]. The former studies the probabilistic nature of user demands (e.g., average workload, utilization, etc.) under the assumption that no permanent structural changes occur due to faults, and is concerned with contention of system resources. The latter studies the probabilistic nature, such as probability of success, mean time to failure, etc., under the assumption of structural changes due to faults.

With the advent of parallel and distributed computer and communication systems, performance evaluation is expanded to encompass concurrency and synchronization aspects, and takes into account of hard/soft deadlines for real time systems. Meanwhile, to incorporate fault-tolerance, reconfiguration, and repair aspects of system behavior, reliability is often evaluated with availability, safety, survivability, and related measures, which are collectively called dependability [47, 63].

As the systems become increasingly complex, an interest of combining performance and dependability evaluations has grown, since performance evaluation often needs to consider the graceful degradation of systems (e.g. fault-tolerance system), which causes blurs with the evaluation of dependability, in particular, the reliability measure. Early investigations on the necessity of combined assessment of performance and dependability for complex computer and communication systems are carried out by [9, 12]. As a result, the "performability", a unifying model for evaluation of performance and reliability on Markov models was proposed by J. F. Meyer [49] in early 1980s. Until now, performance-dependability modeling has been widely accepted for the design and analysis of complex computer and communication systems; this is partly due to the availability of software tools that allows huge underlying Markov models being generated automatically from the abstract high level formalisms.

The classic mechanism for describing performance measures within a high–level model is to use a reward function, that assigns a real–value to each underlying state of the model (see for instance [53] for Petri nets or [18] for process algebra). Performance queries can then be expressed in terms of the expected reward at a fixed time (including infinite time for steady–state), or the expected accumulated reward for a time interval. This idea was later extended to capture path information [55], by utilizing a path automata that is combined with the underlying Markov chain via a synchronous product. A limitation of this work is the lack of a formal language for describing queries or a mechanism for constructing the path automata.

On the other hand, model checking has had huge success in automated formal verification. Having realized that a Morkov model is also a finite state machine, many researches are being developed in applying model checking techniques to Markov models, for path-based performance-dependability analysis. To express dependability related properties, traditional model checking logics such as linear temporal logic (LTL) [58] and computation tree logic (CTL) [20] have evolved into *quantitative* model checking logics; the original temporal logics are extended by adding or modifying operators (e.g., PLTL [22, 30], PCTL [33], pCTL* [3, 11], DCTL [25], CSL [6], asCSL [5], CSL^{TA} [29]), and can express properties such as "with probability at least 0.95, the system will not reach a deadlocked state before time 10". Later, by combing a reward structure with Markov models through a classic mechanism that allows the specification of *reward* functions within high level models, quantitative model checking logics are further extended/adapted to incorporate performance-related measures. Most recent example logics include CSRL [7] and its model checking algorithms [35], DTRMC [2], and some individual rewards-based operators described in [41].

While these quantitative model checking logics are powerful, they are not quite suitable for expressing performance-dependability related measures, since they are limited to producing only true or false responses, precisely because they are *logics* (although in practice, a real–valued response can sometimes be obtained for the outer–most path quantifier). As a concrete example, *survivability* analysis is important in wide variety of applications such as military command, control, communication systems, disaster-based optical networks, etc. [47]; and a typical survivability query can be described as "After an occurrence of failure, what is the probability that the system will reach a set of states under

normal operation?" or "If a model encounters a failure, what is the expected time to reach a recovery state?" To answer these type of complex queries, we identify several challenges. One of them is how to come up with a more general approach towards the unification of model checking and performance evaluation. The other main challenges include how to solve such techniques efficiently, identify the useful performance related queries that cannot be expressed by the existing techniques, and how to demonstrate the practicality of our approach.

For this PhD work, we propose a more general novel mechanism for expressing both performance measures and dependability properties in a single framework. Unlike the recent works [2, 7, 35, 41] (as we mentioned above) which incorporate the performance-related measures into model checking logics (performance measures are ultimately limited by the boolean values), we extend the model checking formalism with real values. As such, we can express not only complex performance-dependability queries such as the survivability measure example that are mentioned above, but also the quantitative model checking queries like "what is the probability for the system to reach a deadlock state before 100 time units?" in formal, exact, succinct fashion, and processed automatically. To summarize, our contributions include: 1). A state based formal language that can take real values as input and output real values, while retaining the expressive power of the existing formal logics. 2). A rigorous comparison to the existing work. 3). An action and state based formal language that distinguishes actions for an input model. 4). A software tool that implements those approaches. 5). Application examples for the illustration of the feasibility of our approach.

The remainder of this document is organized as follows. Chapter 2 presents notations and background information that represents the current state of the art. Chapter 3 presents Markov processes. Chapter 4 presents traditional model checking. Chapter 5 presents probabilistic model checking. These chapters are mainly reviews of the existing work. The following chapters are contributions of this work. Chapter 6 presents real valued formalisms. Chapter 7 presents a rigorous comparison in expressive power with PLTL and PCTL. Chapter 8 presents action and state based formalism that extends the one presented in Chapter 6. Chapter 9 presents a software tool for the real valued formalisms. Chapter 10 presents application examples for the techniques developed in Chapter 6 and 8 and discusses experimental results. Chapter 11 concludes this work and discusses future project.

CHAPTER 2. BACKGROUND

To help understand the extension of probabilistic model checking and probabilistic structure which will be discussed in detail later, in this chapter, we describe basic probability theories [13, 14, 59] that are relevant to this work. In particular, we discuss concepts of measures that allow us to define a quantitative measure for this work later, concepts of random variables and several distributions that are related to the probabilistic structures under consideration.

The following presentation is arranged as follows: Section 2.1 describes notations that are commonly used throughout this work. Section 2.2 recalls the notion of *sigma algebras* and measure. Section 2.3 presents basic definitions of random variables and definition of probability measures. Section **??** describes several probability distributions and their properties.

2.1 Notations

Throughout this work, the basic notations used are *sets*, *matrices*, and *vectors*. For consistency, we use the following symbols and rules to describe these commonly used notations.

Set Notation

In general, a *set* is denoted in upper-case calligraphic letters, except for the following several classic notations.

- $\mathbb{N}$ denotes the set of natural numbers, $\{0, 1, 2, 3, \cdots\}$. It includes 0.
- $\mathbb{R}$ denotes the set of real numbers.
- $\mathbb{R}^+$ denotes the set of positive real numbers.
- $\mathbb{R}^*$ denotes the set of nonnegative real numbers.

Note that all other sets are denoted in upper-case calligraphic letters. In particular, $\mathcal{S}$ denotes the set of states (i.e. nodes) for a given graph, such as a probabilistic model as shown in Figure 2.1 (where the values on all arcs are probabilities). Unless otherwise specified (e.g., every state has a meaningful description in a clear context), states are indexed by a sequence of natural numbers such as $\{0, 1, \cdots, |\mathcal{S}| - 1\}$, where $|\mathcal{S}|$ denotes the cardinality of $\mathcal{S}$.

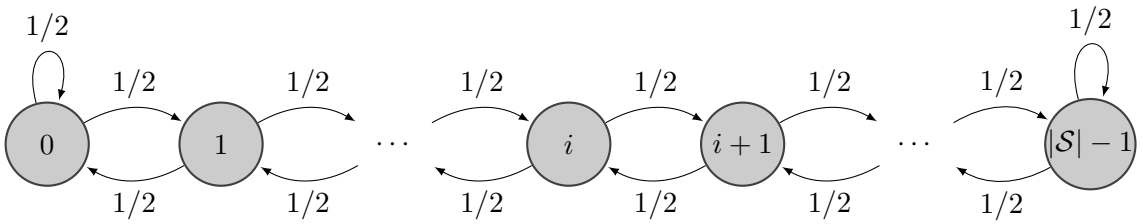


Figure 2.1: An Example of Probabilistic Model, with $\mathcal{S} = \{0, 1, \cdots, |\mathcal{S}| - 1\}$.

Matrix Notation

In general, matrices are written in upper-case **bold** letters, such as $\mathbf{R}$, $\mathbf{N}$, etc.. The following lists several concrete notations.

- $\mathbf{R}^{m \times n}$ denotes a real-valued matrix that has dimension of m rows and n columns; when the context is clear, it is simply written as $\mathbf{R}$.
- Given a square matrix $\mathbf{R}$, the *inverse* of $\mathbf{R}$ is denoted by $\mathbf{R}^{-1}$.
- Given a matrix $\mathbf{R}$, the *transpose* of $\mathbf{R}$ is denoted by $\mathbf{R}^{\mathrm{T}}$.
- Given a matrix $\mathbf{R}$, $\mathbf{R}[i, j]$ denotes an element at row i and column j.
- $\mathbf{I}$ denotes an *identity* matrix of dimension $n \times n$, assuming size n is known in a given context.
- $\mathbf{1}$ and $\mathbf{0}$ denote the matrices of all ones and zeroes with m rows and n columns, respectively, assuming the dimension of $m \times n$ is known in the a given context.
- A submatrix of $\mathbf{R}$ is denoted by $\mathbf{R}[\mathcal{I}, \mathcal{J}]$ with rows $\mathcal{I}$ and columns $\mathcal{J}$.

Vector Notation

In general, vectors are denoted by lower-case bold letters, such as $\boldsymbol{\pi}$, $\mathbf{h}$, $\mathbf{f}$, etc..

- Given a vector $\boldsymbol{\pi}$, $\boldsymbol{\pi}[i]$ denotes an element of $\boldsymbol{\pi}$.
- Given a vector $\boldsymbol{\pi}$, $\boldsymbol{\pi}[\mathcal{I}]$ denotes a sub-vector of $\boldsymbol{\pi}$, with $\mathcal{I} \subseteq \mathcal{S}$.
- $Diag(\mathbf{f})$ denotes a square matrix with $\mathbf{f}$ along the diagonal and zeroes elsewhere,

2.2 σ-Algebras and Measure Theory

Definition 1 (σ-algebra)**.** *A σ-algebra [21] is a subset Σ of the power set of a set $\mathcal{X}$ such that:*

1. *$\mathcal{X}, \emptyset \in \Sigma$.*
2. *if $\mathcal{C} \in \Sigma$, then $\mathcal{X} \setminus \mathcal{C} \in \Sigma$. That is, Σ is closed under complementation.*
3. *if $\mathcal{C}_0, \mathcal{C}_1, \ldots$ with $\mathcal{C}_i \in \Sigma$, then $\bigcup_{i=0}^{\infty} \mathcal{C}_i \in \Sigma$ and $\bigcap_{i=0}^{\infty} \mathcal{C}_i \in \Sigma$. That is, a σ-algebra Σ is closed under countable sets of operations (σ often means countable in mathematics).*

There may be many σ-algebras over set $\mathcal{X}$. Let $\mathfrak{F}$ denote a family (i.e., nonempty collection) of subsets of $\mathcal{X}$, let $\mathcal{A}_{\mathfrak{F}}$ denote the collection of all σ-algebras that contains $\mathfrak{F}$, then by taking intersections of all elements in $\mathcal{A}_{\mathfrak{F}}$, we obtain the smallest σ-algebra that contains $\mathfrak{F}$, called the σ-algebra *generated* by $\mathfrak{F}$. If $\mathcal{X} = \mathbb{R}$, then the σ-algebra generated by the collection of open subsets of $\mathcal{X}$, $\{(a, b) : a, b \in \mathbb{R}\}$ (or equivalently the collection of closed subsets of $\mathcal{X}$, $\{[a, b] : a, b \in \mathbb{R}\}$), is called *Borel* σ-algebra. For more rigorous discussions on Borel σ-algebra, we refer the interested reader to [21].

Note that elements of an arbitrary σ-algebra Σ are called measurable sets. A function $\mu : \Sigma \to [0, \infty]$ is a *measure* if $\mu(\emptyset) = 0$, and for any sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots$ of disjoint sets in Σ,

$$\mu\left(\bigcup_{i=0}^{\infty} \mathcal{C}_i\right) = \sum_{i=0}^{\infty} \mu(\mathcal{C}_i). \tag{2.1}$$

2.3 Random Variables and Probability Measures

As per the probability theory [13, 14, 59], let $(\mathcal{X}, \Sigma)$ be a measurable space, where $\mathcal{X}$ represents a sample space (i.e. the set of all possible outcomes) and Σ is a subset of the power set of $\mathcal{X}$, then a *random variable*, denoted by X, is a function, that maps the sample space $\mathcal{X}$ to some set, denoted by $\mathcal{S}$. If $\mathcal{S} = \mathbb{R}$, then X is a *real valued* random variable; in this case, we have:

$$X : \mathcal{X} \to \mathbb{R} \text{ such that the set } \{\omega \mid X(\omega) \leq x\} \in \Sigma \text{ for every } x \in \mathbb{R},$$

where x is called a *realization* of X, and $\{\omega \mid X(\omega) \leq x\}$ denotes the set of outcomes each of which has a realization value less than or equal to x. For this work,we are interested in the real valued random variables, which are simply called random variables, for all subsequent discussions.

If $\mathcal{S}$ is countable with $\mathcal{S} \subseteq \mathbb{N}$, then X is a *discrete random variable*. Examples of discrete random variables include the number of defective items in a box of twenty cell phones of the same brand and model, the number of customers in a restaurant at dinner time, etc. If $\mathcal{S}$ is uncountable with $\mathcal{S} \subseteq \mathbb{R}$, then X is a *continuous random variable*. Examples of continuous random variables include the time required to drive three miles, the amount of fat in one pound of pork, etc. Note that by convention, random variables are denoted by capital letters.

As an example, consider a sequence of $n \in \mathbb{N}$ consecutive coin tosses. An appropriate sample space is $\mathcal{X} = \{0, 1\}^n$, where 1 stands for tails and 0 for heads. Let Σ be the collection of all subsets of $\mathcal{X}$, we are interested in the "number of tails" obtained in this experiment. Let $\omega = (\omega_1, \cdots, \omega_n)$ denote the outcome of a specific experiment, where ω_i is either 0 or 1, with $1 \leq i \leq n$, then the quantity of the number of tails can be described by the random variable $X : \mathcal{X} \to \mathbb{N}$, defined by $X(\omega) = \omega_1 + \cdots + \omega_n$. Under this scenario, the set $\{\omega \mid X(\omega) < 3\}$ is simply the *event* (i.e., the set of outcomes of an experiment) saying that there are fewer than 3 tails overall, that belongs to the set Σ.

We are now ready to define probability measure. Let $\mathcal{X}$ be a sample space, let Σ be an associated σ-algebra, then a *probability measure*, denoted by $\Pr$, is a function, defined by $\Pr : \Sigma \to [0, 1]$, and it must satisfy the following axioms:

1. $0 \leq \Pr(\mathcal{C}) \leq 1$, for all $\mathcal{C} \in \Sigma$,

2. $\Pr(\emptyset) = 0$; and $\Pr(\mathcal{X}) = 1$ (i.e., the probability measure of the whole space is 1).

3. if $\mathcal{C}_0, \mathcal{C}_1, \ldots \in \Sigma$ are pairwise disjoint, then $\Pr(\bigcup_{i=0}^{\infty} \mathcal{C}_i) = \Sigma_{i=1}^{\infty} \Pr(\mathcal{C}_i)$,

Notice that a probability measure is indeed a measure μ, except the codomain is limited to the range of $[0, 1]$, rather than $\mathbb{R}$.

Given the definition of probability measure, a *conditional probability* measures the probability of an event given that another event has occurred. Let A, B describe the events, then the conditional probability is defined as:

$$\Pr(B \mid A) = \frac{\Pr(A \cap B)}{\Pr(A)} \tag{2.2}$$

with $\Pr(A) > 0$. The following discusses basic notions and properties related to discrete random variables and continuous random variables, respectively.

Given a discrete random variable X, the probability for a *realization* of X is denoted by $\Pr(X = i)$, and the *probability distribution* of X is a list of probabilities associated with each of its possible value. In fact, a discrete random variable X is associated with *probability mass function* (pmf) with the following properties:

$$(i) \quad 0 \leq \Pr(X = i) \leq 1 \qquad \text{for all } i \in \mathcal{S}$$

$$(ii) \quad \textstyle\sum_{i \in \mathcal{S}} \Pr(X = i) = 1$$

For example, an unbiased dice roll can take six values numbered as $1, 2, 3, 4, 5, 6$; the probabilities associated with each outcome is $\frac{1}{6}$. Then,

- The probability that the variable X is equal to 1 is: $\Pr(X = 1) = \frac{1}{6}$.
- The probability that the variable X is equal to 1 or 2 is: $\Pr(X = 1 \text{ or } X = 2) = \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$.
- The probability that the variable X is greater than or equal to 3 is: $\Pr(X \geq 3) = 1 - \Pr(X = 1 \text{ or } X = 2) = \frac{2}{3}$.

The *expected value* of a discrete random variable X, denoted by $E(X)$, refers to the long-run average value of repetitions of the experiment it represents; precisely, it is the probability-weighted

average of all possible values [13], and can be defined as

$$E(X) = \sum_{x} x \Pr(X = x) \tag{2.3}$$

Take the dice roll example as described above, the expected value of a dice roll is $(1 * \frac{1}{6} + 2 * \frac{1}{6} + 3 * \frac{1}{6} + 4 * \frac{1}{6} + 5 * \frac{1}{6} + 6 * \frac{1}{6}) = 3.5$.

Then, the *conditional expectation* [57] of a discrete random variable B given an event A, denoted by $E(B|A)$, is the expectation of B under the conditional distribution (i.e. probability distribution for a sub sample space) given A and can be defined as

$$E(B \mid A) = \sum_{all\, b} b \cdot \Pr(B = b \mid A) \tag{2.4}$$

$$= \frac{\sum_{all\, b} b \cdot \Pr(B = b \cap A)}{\Pr(A)} \tag{2.5}$$

In contrast, a continuous random variable X is characterized by a *cumulative distribution function*, abbreviated as *cdf*, which is typically denoted by upper-case $F(x)$, i.e., $F(x) = \Pr\{X \leq x\}$ for every value $x \in \mathbb{R}$. Formally, $F(x)$ for a continuous random variable X has the following properties:

(i)	$0 \leq F(x) \leq 1$	for all $x \in \mathbb{R}$
(ii)	$F(x_1) \leq F(x_2)$	if $x_1 \leq x_2$
(iii)	$\lim_{x \to -\infty} F(x) = 0$ and $\lim_{x \to \infty} F(x) = 1$	

The derivative of the distribution function $F(x)$ is called the *density* function, denoted by lower case $f(x)$, with the following properties:

(i)	$f(x) \geq 0$	for all $x \in \mathbb{R}$
(ii)	$\int_{-\infty}^{\infty} f(x)dx = 1$	

By properties of $F(x)$ and $f(x)$, it follows that:

$$(i) \qquad F(x) = \Pr(X \leq x) = \int_{-\infty}^{x} f(y)dy$$

$$(ii) \qquad \Pr(a \leq X \leq b) = \int_{a}^{b} f(x)dx$$

Given a continuous random variable X with probability density function $f(x)$, the expected value of $E(X)$ can be defined as:

$$E(X) = \int_{-\infty}^{+\infty} xf(x)dx$$

CHAPTER 3. MARKOV PROCESSES

Low-level modeling formalisms such as Markov chains have long been accepted for the analysis of performance measures. Due to its large state space typically involved, however, manually constructing and analyzing such a model often impose a big challenge. With the advent of software tools such as GreatSPN [4] and SMART [16] in more recent years, large-scaled Markov chains can now be generated automatically from high-level modeling formalisms such as stochastic Petri net (SPN). Meanwhile, as a result of great success in the area of model checking, automated analysis towards the unification of performance measures and formal verification making researching in numerical analysis of such models again be very much alive.

In this chapter, we first give general concepts of random processes. Then we discuss details about a special type of the random processes, namely, discrete time Markov chains, that serve as a fundamental type of models in combining with model checking techniques. Along with the properties and concepts about this type of random process, we also discuss several real-valued measures that inspired this work. Finally, for future extension of this work, we give a very brief introductory description of several other types of random processes (such as continuous time Markov chain). Note that for the following discussion on the stochastic processes, interested readers should refer to [13, 38, 60] for more details about the topic; [50] also has some inspiring examples.

3.1 Stochastic Processes and Markov Chains

Definition 2 (stochastic process)**.** *A stochastic (random) process is a sequence of random variables* $\{X(t) : t \in \mathcal{T}\}$*, where* $\mathcal{T}$ *is a time related parameter set .*

If $\mathcal{T}$ is countable, then $\{X(t)\}$ is a *discrete time* stochastic process; otherwise, it is a *continuous time* stochastic process. The *state space*, $\mathcal{S}$, is the set of all possible values of $X(t)$. Like $\mathcal{T}$, $\mathcal{S}$ can be

continuous or discrete as well. So, depending on whether or not $\mathcal{S}$ and $\mathcal{T}$ are discrete or continuous, a stochastic process can be classified into four types. For example, if $\{X(t)\}$ represents the outcome of the t^{th} toss of a fair dice, then $\{X(t), t \geq 1\}$ is a discrete time discrete state random process, where $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$. If $X(t)$ describes temperatures at the end of t^{th} hour of a day, then it is a discrete time continuous state random process. If $X(t)$ describes temperatures at any time of a day, then it is a continuous time continuous state random process. If $X(t)$ describes the number of jobs in a queue at time t, then typically it is a continuous time discrete state random process.

Definition 3 (Markov process)**.** *A Markov process is a special case of stochastic process that satisfies the* Markov *property, or so called "memoryless" property, which is defined as:*

$$\Pr\{X(t_{n+1}) = s_{n+1} \mid X(t_n) = s_n, \ldots, X(t_0) = s_0\} = \Pr\{X(t_{n+1}) = s_{n+1} \mid X(t_n) = s_n\}$$

where $n \in \mathbb{N},\ s_0, \ldots, s_{n+1} \in \mathcal{S},\ t_0,\ \ldots, t_{n+1} \in \mathcal{T}$ *and* $t_0 < \ldots < t_{n+1}$.

Simply put, the definition says that "the future depends only on the current state, not the past." In terms of performance measures, the memoryless property is the key for *transient analysis* and *steady-state analysis*, which are considered as the basis for the quantitative evaluation of the behavior of a stochastic process representing the time-evolution of a discrete-state event-driven dynamic system (DEDS) [8]. Performance related measure will be discussed in more detail as we reach each specific type of Markov processes.

Definition 4 (Markov chain)**.** *A Markov process with a discrete state space* $\mathcal{S}$ *is called a Markov chain.*

For this work, we are only interested in the type of stochastic processes that are Markov chains.

3.2 Discrete-Time Markov chain

Definition 5 (discrete-time Markov chain)**.** *A discrete-time Markov chain (DTMC) is a Markov chain with a discrete set of time indices. With* $\mathcal{T} \equiv \mathbb{N}$*, a DTMC can be simply written as* $\{X(n) : n \in \mathbb{N}\}$.

Specifically, a DTMC consists of the following three components:

- *State space $\mathcal{S}$.* $\mathcal{S} = \{0, 1, 2, \cdots\}$ is a finite (or countably infinite) set of states that the random variables X may take on.

- *Probability transition rule.* This is specified by $|\mathcal{S}| \times |\mathcal{S}|$ transitional matrix $\mathbf{P}$. Note that for this work, we assume the Markov chains are *time-homogeneous*, meaning the transition probabilities are independent of time n and are defined by:

$$\Pr\{X(n+1) = s_j \mid X(n) = s_i\} = \Pr\{X(n) = s_j \mid X(n-1) = s_i\}, \text{ for } n > 0.$$

  By the definition of time-homogeneous Markov chains, we have $\mathbf{P} = \mathbf{P}_n$, where $\mathbf{P}_n$ means the probability transition matrix at time step n, and $\mathbf{P}_n[i, j]$ is the conditional probability that the chain jumps to the state j at time step n given that the chain is in state i at time step $n - 1$. That is, $\mathbf{P}_n[i, j] = \Pr\{X(n) = j \mid X(n-1) = i\}$. For example,

$$\Pr\{X(222) = 5 \mid X(221) = 3\} = \Pr\{X(66) = 5 \mid X(65) = 3\},$$

$$\Pr\{X(2) = 3 \mid X(1) = 1\} = \Pr\{X(6) = 3 \mid X(5) = 1\}.$$

  Note that $\mathbf{P}_n$ is different from $\mathbf{P}^n$ which means the probability matrix to the power of n.

- *Initial distribution vector $\boldsymbol{\pi}_0$.* This is the probability distribution of the Markov chain at time 0. For each state $i \in \mathcal{S}$, we denote by $\boldsymbol{\pi}_0[i]$ the probability $\Pr\{X(0) = i\}$ that the Markov chain starts out in state i. Formally, $\boldsymbol{\pi}_0$ obeys the following rule:

$$\begin{aligned} &(i) \quad \boldsymbol{\pi}_0[i] \in [0, 1] \text{ for all } i \in \mathcal{S}, \text{ and} \\ &(ii) \quad \textstyle\sum_{i \in \mathcal{S}} \boldsymbol{\pi}_0[i] = 1. \end{aligned}$$

Now lets look at example of DTMC and see how it can be used to model a real life situation. A dreamer from a poor family believes that luck will turn around year after year. So she came up with the following Markov chain model, trying to predict how many years that she will get the best of luck and get rid of being poor. The model has three states: *poor*, *middle* (for middle class), and *rich*. If a person is poor, then there is a 0.5 probability to become middle class; there is 0.4 probability that she will stay

being poor, and there is 0.1 probability that she will become rich. If a person is in middle class, then she's likely to become poor with 0.3 probability, and stay being middle class with 0.5 probability, and 0.2 probability to become rich. If a person is rich, then she has 0.5 probability keeping in rich status, and 0.4 probability to become middle class, and 0.1 probability to become poor. The scenario is shown in Figure 3.1.

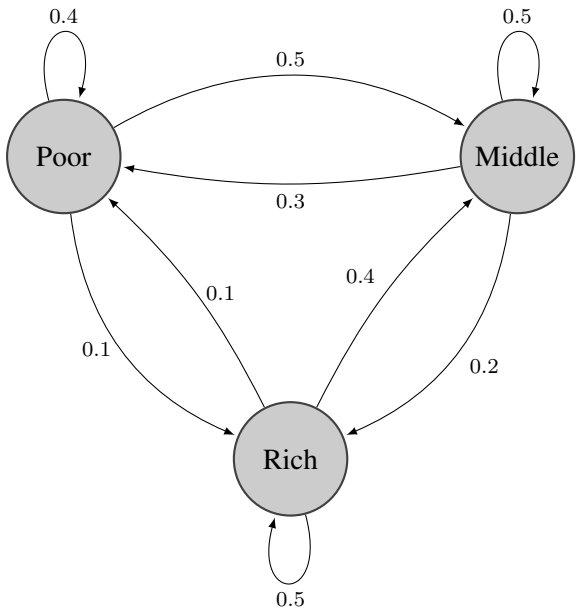


Figure 3.1: Luck of Fortune DTMC

Now before we can answer questions such as "Given a person is poor, what is the probability that she will be in middle class after 3 years?" on the "Luck of Fortune" model, we need to describe the following analysis technique.

3.2.1 Transient Analysis

Given a DTMC, the *transient analysis* can handle questions like: If the process is in state i at time 0, what is the probability distribution of states at time n, for $n > 0$? If $n = 1$, then this can be expressed as:

$$\text{What is } \Pr\{X(1) = j \mid X(0) = i\}, \text{ for all state } j?$$

The solution is simply equal to the row i of $\mathbf{P}$, i.e., $\boldsymbol{\pi}_1[j] = \mathbf{P}[i, j]$, for all $j \in \mathcal{S}$. Similarly, we may also ask: given a distribution of states at time 0, what is the probability distribution of states at time 2, or 3, $\cdots$.

To generalize, let vector $\boldsymbol{\pi}_n$ denote the distribution of the chain at time n; given the probability transition rule of a homogeneous DTMC, by the law of total probability, the probability distribution at time $n+1$, denoted by $\boldsymbol{\pi}_{n+1}$ can be computed as follows:

$$\begin{aligned}\boldsymbol{\pi}_{n+1}[j] &= \Pr\{X(n+1) = j\} \\ &= \sum_{i \in \mathcal{S}} \Pr\{X(n) = i\} \Pr\{X(n+1) = j \mid X(n) = i\} \\ &= \sum_{i \in \mathcal{S}} \boldsymbol{\pi}_{\mathbf{n}}[i] \mathbf{P}[i, j]\end{aligned}$$

which, in matrix notation, is the equation

$$\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \mathbf{P}. \tag{3.1}$$

Alternatively, this can be written in terms of initial distribution

$$\boldsymbol{\pi}_n = \boldsymbol{\pi}_0 \mathbf{P}^n. \tag{3.2}$$

Now take the example of "Luck of Fortune", suppose the initial distribution is $\boldsymbol{\pi}_0 = [1,\ 0,\ 0]$. Given

$\mathbf{P} =$

	$Poor$	$Middle$	$Rich$
$Poor$	0.4	0.5	0.1
$Middle$	0.3	0.5	0.2
$Rich$	0.5	0.1	0.4

Then the answer to the question of "What is the luck of the dreamer be in the middle class in five years?" can be analyzed as follows:

- $\boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 \mathbf{P} = [0.40000,\ 0.50000,\ 0.10000]$

- $\boldsymbol{\pi}_2 = \boldsymbol{\pi}_1\mathbf{P} = [0.36000,\ 0.46000,\ 0.18000]$
- $\pi_3 = \pi_2\mathbf{P} = [0.37200,\ 0.42800,\ 0.20000]$
- $\pi_4 = \pi_2\mathbf{P} = [0.37720,\ 0.42000,\ 0.20280]$
- $\pi_5 = \pi_2\mathbf{P} = [0.37828,\ 0.41888,\ 0.20284]$. ✓

The solution says that, in five years, the dreamer is likely to stay in poor with probability 0.37828, is like be in middle class with probability 0.41888, and is likely to become rich with probability 0.20284.

3.2.2 Recurrent, Transient, Irreducible Properties

Definition 6 ($i \rightsquigarrow j$)**.** *We say a state j is* reachable *from state i and written by $i \rightsquigarrow j$ if $\exists n > 0, n \in \mathbb{N}, \Pr\{X(n) = j \mid X(0) = i\} > 0$; otherwise, we write $i \not\rightsquigarrow j$.*

The definition says that starting from state i now, it is possible to go to state j at some time n in the future. If no such n exists, then state j is not reachable from state i. Given a DTMC, state i is reachable from state j if and only if there is a *path* in the DTMC graph between the two states. Consider Figure 3.2 for an example, we have state e reachable from states a, b, c, d, and e; state f is reachable from states a, b, c, d, and f; state a is reachable from states b, c, d; state b is reachable from states a, c, d; state c is reachable from states a, b, d; and state d is reachable from states a, b, c. But states a, b, c, d, f are not reachable from state e, neither are states a, b, c, d, e reachable from state f.

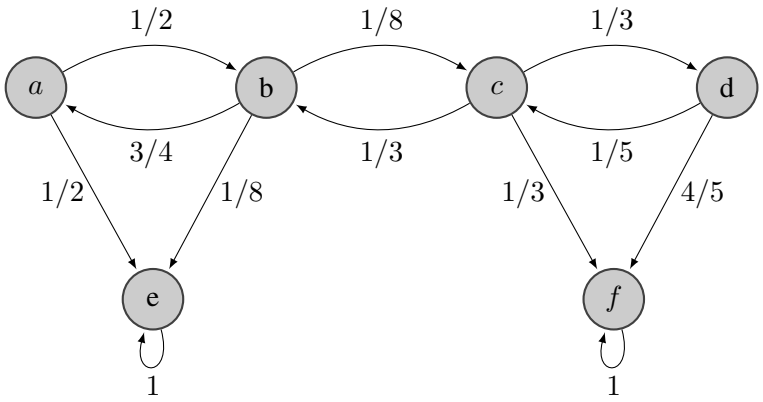


Figure 3.2: Another DTMC Example

Definition 7 (mutually reachable)**.** *If $i \rightsquigarrow j$ and $j \rightsquigarrow i$, then states i and j are* mutually reachable.

Definition 8 (transient state)**.** *A state i is called* transient *if there exists a state j such that $i \rightsquigarrow j$ and $j \not\rightsquigarrow i$.*

In the above example, states a, b, c and d are transient, because they can reach states e and f, but not vice versa. Obviously, if a state i is transient, then this means there is a non-zero probability that the DTMC never return to state i after leaving, and as time goes to infinity, the probability of being in state i goes to zero.

Definition 9 (recurrent state)**.** *A state i is recurrent if starting in state i, the Markov chain will, with probability* 1*, eventually return to the state.*

In the above example, states e and f are recurrent. In the "Luck of Fortune" example, every state is recurrent. Also, a recurrent state has infinitely many *return times*, which will be visited again later.

Based on the above definitions, we can now partition the state space $\mathcal{S}$ of a Markov chain into the following two types of classes: One is called **recurrent class**, within which all states are mutually reachable, and once a DTMC enters a recurrent class, it remains in that recurrent class forever. Another is called **transient class**, within which all states are transient. Figure 3.3 shows an example partition of recurrent class and transition class, in which the set of states $\{c, d, f\}$ and the set of state $\{e\}$ are recurrent classes, and $\{a, b\}$ is a transient class. For more rigorous discussions on recurrent classes, we refer readers to [60].

Definition 10 (absorbing state)**.** *A state i is absorbing if it can reach only itself, i.e. the state has no outgoing arcs leading to other states, but only has a self loop arc with probability one.*

In the above example, states e and f are absorbing. Obviously, every absorbing state is also a recurrent state.

Definition 11 (irreducible Markov chain)**.** *A Markov chain is said to be irreducible if its state space $\mathcal{S}$ is a recurrent class, i.e., all states are mutually reachable; otherwise it is reducible.*

For example, the DTMC shown in Figure 3.3 is reducible, because $\mathcal{S}$ is not a recurrent class. In contrast, the "Luck of Fortune" DTMC example is irreducible.

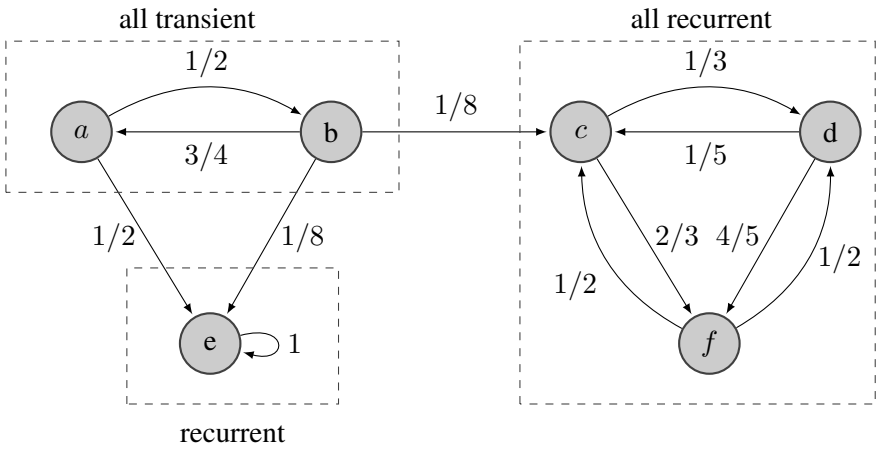


Figure 3.3: Classification of DTMC States

Definition 12 (periodicity)**.** *A recurrent state i is said to have period d_i if any return to state i must occur in multiples of d_i time steps. Let* gcd *denote the greatest common devisor, then the period of a state in DTMC is defined by:*

$$d_i = \gcd\{n : \mathbf{P}^n[i,i] > 0\}.$$

where n is a possible return time to state i, and there will be infinitely many return times for a recurrent state i.

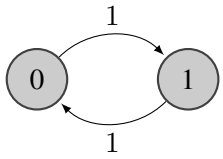


Figure 3.4: Recurrent DTMC with Period $d = 2$.

Take the example shown in Figure 3.4, when starting from state 0, it revisits 0 when $n = 2$, $n = 4$, $n = 6$, $\cdots$, so the period is 2. In fact, the possible returning time to both state 0 and state 1 are: $\{2, 4, 6, 8, 10, \cdots\}$, so $d_0 = d_1 = 2$. In this case, if we start with the initial distribution, say $\boldsymbol{\pi}_0 = [1, 0]$, then $\boldsymbol{\pi}_n$ and $\boldsymbol{\pi}_{n+1}$ alternates between $[1, 0]$ and $[0, 1]$.

Definition 13 (aperiodic Markov chain)**.** *An irreducible DTMC is said to be* aperiodic *if there exists a state i with period d_i equal to* 1*, and periodic otherwise.*

Intuitively, given an irreducible DTMC, if there exists a state i such that $\mathbf{P}[i,i] > 0$, then the DTMC would be aperiodic, because the set of returning time for i can be $\{1, 2, 3, \cdots\}$, so the gcd for i is $d_i = 1$.

Definition 14 (ergodic Markov chain)**.** *An irreducible, aperiodic Markov chain (i.e., its states are all aperiodic) is called ergodic Markov chain.*

Consider the "Luck of Fortune" DTMC, shown in 3.1, it is irreducible and aperiodic, so the "Luck of Fortune" DTMC is an ergodic Markov chain. Given the above definitions and properties, we are now ready to describe several important performance measures for DTMCs.

3.2.3 Absorbing DTMCs

Definition 15 (absorbing DTMC)**.** *An absorbing DTMC is defined by the set of states each of which is either a transient or an absorbing state, i.e., $\mathcal{S} = \mathcal{S}_z \cup \mathcal{S}_a$, where $\mathcal{S}_z$ is a set of transient states, and $\mathcal{S}_a$ is a set of absorbing states.*

By definition, the transition probability matrix $\mathbf{P}$ of an absorbing DTMC can be rearranged into the following block structure:

$$\mathbf{P} = \left[\begin{array}{c|c} \mathbf{P}[\mathcal{S}_z,\, \mathcal{S}_z] & \mathbf{P}[\mathcal{S}_z,\, \mathcal{S}_a] \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right] \tag{3.3}$$

where $\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]$ are the arcs from transient states to transient states, $\mathbf{P}[\mathcal{S}_z, \mathcal{S}_a]$ are the arcs from transient states to absorbing states, the block of $\mathbf{0}$ indicates no actual arcs from absorbing states to transient states, and the identity matrix $\mathbf{I}$ (with dimension $|\mathcal{S}_a \times \mathcal{S}_a|$) contains the self-loop arcs from absorbing to absorbing states only.

To this end, we discuss two types of performance measures that are tied to absorbing DTMCs: *mean time to absorption* and *limiting distributions*. The former answers questions like "In the long run, what is the average time that the DTMC takes to reach an absorbing state given an initial distribution?" The latter answer questions like "In the long run, what is the probability that the DTMC eventually reached each of the absorbing states?" Before we answer these questions, let's look at several properties followed by a fundamental theorem of associated with an absorbing DTMC. [13, 38, 50, 60] give more detailed discussion about those properties and theorems.

Property 16. $\mathbf{P}^n[\mathcal{S}_z, \mathcal{S}_z] = \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^n$.

Proof. By induction on n: If $n = 1$, then the equation holds trivially. For $n > 1$, suppose $\mathbf{P}^{n-1}[\mathcal{S}_z, \mathcal{S}_z] = \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^{n-1}$ holds. Then

$$\begin{aligned}
\mathbf{P}^n[\mathcal{S}_z, \mathcal{S}_z] &= \mathbf{P}^{n-1}\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] \\
&= \mathbf{P}^{n-1}[\mathcal{S}_z, \mathcal{S}_z]\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] + \mathbf{P}^{n-1}[\mathcal{S}_z, \mathcal{S}_a]\mathbf{0} \\
&= \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^{n-1}\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] \\
&= \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^n.
\end{aligned}$$

□

Property 17. *Let* $\mathbf{M}$ *be a square matrix with non-negative elements, with largest rowsum* $\alpha < 1$.

$$\lim_{n\to\infty} \mathbf{M}^n = \mathbf{0}.$$

Proof. First, we show that the largest rowsum in $\mathbf{M}^n$ is at most α^n, for $1 < n \in \mathbb{N}$. We use proof by induction. Base case: $n = 1$, in this case, we have $\mathbf{M}^n = \mathbf{M}^1 = \mathbf{M}$, so it trivially holds that the largest rowsum in $\mathbf{M}$ is at most α. Inductive case: Let $r \times r$ denote the dimension of rows and columns of the square matrix $\mathbf{M}$. Let $\alpha_1, \cdots, \alpha_r$, denote the rowsum value for each row in $\mathbf{M}$, respectively, with $\alpha = max\{\alpha_1, \cdots, \alpha_r\}$. We assume for any $n > 2$, it is true that the largest rowsum in $\mathbf{M}^n$ is at most α^n. We show the largest rowsum in $\mathbf{M}^{n+1}$ is at most α^{n+1}. For convenience, let $\mathbf{M}'$ denote the matrix

for $\mathbf{M}^n$. Let β_i^n denote the rowsum value in $\mathbf{M}'$ corresponding to the i^{th} row. Then,

$$\begin{aligned}
\beta_i^{n+1} &= \mathbf{M}'[i,1]\cdot\mathbf{M}[1,1] + \cdots + \mathbf{M}'[i,r]\cdot\mathbf{M}[r,1] \\
&+ \mathbf{M}'[i,1]\cdot\mathbf{M}[1,2] + \cdots + \mathbf{M}'[i,r]\cdot\mathbf{M}[r,2] \\
&\cdots \\
&+ \mathbf{M}'[i,1]\cdot\mathbf{M}[1,r] + \cdots + \mathbf{M}'[i,r]\cdot\mathbf{M}[r,r] \\
&= \mathbf{M}'[i,1]\cdot(\mathbf{M}[1,1] + \cdots + \mathbf{M}[1,r]) + \cdots + \mathbf{M}'[i,r]\cdot(\mathbf{M}[r,1] + \cdots + \mathbf{M}[r,r]) \\
&= \mathbf{M}'[i,1]\cdot\alpha_1 + \cdots + \mathbf{M}'[r,1]\cdot\alpha_r \\
&\leq (\mathbf{M}'[i,1] + \cdots \mathbf{M}'[r,1])\cdot\alpha \\
&\leq \alpha^n\cdot\alpha = \alpha^{n+1}
\end{aligned}$$

We now have proved that for any $n > 1, n \in \mathbb{N}$, the largest rowsum in $\mathbf{M}^n$ is at most α^n. Since $\alpha < 1$, α^n goes to 0 if n goes to infinity. Therefore, all the rowsums in $\mathbf{M}^n$ go to 0 if n goes to infinity, which implies all the elements in $\mathbf{M}^n$ goes to zero, thus $\lim_{n\to\infty}\mathbf{M}^n = \mathbf{0}$.

□

Property 18.

$$\lim_{n\to\infty}\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z]^n = 0$$

Proof. From the absorbing DTMC block structure, we know every row of $\mathbf{P}^m[\mathcal{S}_z,\mathcal{S}_a]$ contains at least one non-zero entry for some finite m, since each transient state will eventually reach an absorbing state. As such, every row of $\mathbf{P}^m[\mathcal{S}_z,\mathcal{S}_a] = \mathbf{P}[\mathcal{S}_z,\mathcal{S}_a]^m$ sums to less than one. Therefore, by property 17, we have

$$\lim_{n\to\infty}\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z]^n = \lim_{n\to\infty}(\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z]^m)^n = \mathbf{0}.$$

□

This is to say, in the long run, the probability that the DTMC stay in all transient states goes to zero, and the probability that the DTMC remain in absorbing states goes to one. That is, we have:

$\lim_{n\to\infty} \Pr\{X_n \in \mathcal{S}_z\} = 0$, and $\lim_{n\to\infty} \Pr\{X_n \in \mathcal{S}_a\} = 1$.

Theorem 19 (fundamental matrix). *Let* $\mathbf{N}$ *denote the fundamental matrix [38, 60] for a given absorbing DTMC, then*

$$\mathbf{N} = (\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z])^{-1} = \sum_{k=0}^{\infty} \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^k$$

Note that $\mathbf{N}[i, j]$ denotes the *expected (total) number of visits to state* j*, given the DTMC is starts in state* i *at time* 0, where $i,\ j$ are transient states. Consider the example shown in Figure 3.2. We have $\mathcal{S}_z = \{a,\ b,\ c,\ d\}$, and $\mathcal{S}_a = \{e,\ f\}$. The transition probability matrix is:

$$\mathbf{P} = \left[\begin{array}{c|cccc|cc} & a & b & c & d & e & f \\ \hline a & 0 & 1/2 & 0 & 0 & 1/2 & 0 \\ b & 3/4 & 0 & 1/8 & 0 & 1/8 & 0 \\ c & 0 & 1/3 & 0 & 1/3 & 0 & 1/3 \\ d & 0 & 0 & 1/5 & 0 & 0 & 4/5 \\ \hline e & 0 & 0 & 0 & 0 & 1 & 0 \\ f & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

The fundamental matrix is:

$$\mathbf{N} = (\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z])^{-1} = \left[\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & -1/2 & 0 & 0 \\ b & -3/4 & 1 & -1/8 & 0 \\ c & 0 & -1/3 & 1 & -1/3 \\ d & 0 & 0 & -1/5 & 1 \end{array}\right]^{-1}$$

Plug in the above matrix (before the inverse) into a matrix calculator, we get:

$$\mathbf{N} = \left[\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1.6461 & 0.8615 & 0.1154 & 0.0385 \\ b & 1.2923 & 1.7231 & 0.2308 & 0.0769 \\ c & 0.4615 & 0.6153 & 1.1538 & 0.3846 \\ d & 0.0923 & 0.1231 & 0.2308 & 1.0769 \end{array}\right]$$

The fundamental matrix says that if the DTMC starts in state a, then the expected number of visits to state a is 1.6461; if it starts in state d, then the expected number of visits to state b is 0.1231, so on and so forth.

One problem in obtaining $\mathbf{N}$ is that though in principle, the inverse of $\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]$ is computable, in practice, the computation is very difficult in the perspective of both storage and efficiency, because even though the DTMC might be *sparse* $\mathbf{N}$ is usually a dense matrix which could pose storage issue, and due to the typically very big size of $\mathbf{I}-\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]$, direct computation of its inverse can be inefficient. Fortunately, there exists a way to avoid direct computation of the inverse for the relevant performance measures which we will be describing below, followed by a practical method of computation.

Property 20 (mean time to absorption)**.** *Given an initial probability distribution $\boldsymbol{\pi}_0$ for an absorbing DTMC, the measure of mean time to absorption can be computed by summing over the elements of vector $\mathbf{m}$, with*

$$\mathbf{m}[j] = \sum_{i \in \mathcal{S}_z} \mathbf{N}[i, j] \cdot \boldsymbol{\pi}_0[i] = [\mathcal{S}_z, j] \cdot \boldsymbol{\pi}_0[\mathcal{S}_z],$$

which denotes the expected number of visits to state j given $\boldsymbol{\pi}_0$. Hence,

$$\mathbf{m} = \boldsymbol{\pi}_0[\mathcal{S}_z] \cdot \mathbf{N} = \sum_{k=0}^{\infty} \boldsymbol{\pi}_k[\mathcal{S}_z],$$

where $\boldsymbol{\pi}_k$ is the probability distribution at time k.

As an example, consider Figure 3.2 again. Suppose $\boldsymbol{\pi}_0 = [1/4,\ 0,\ 1/4,\ 0,\ 1/2,\ 0]$, then $\boldsymbol{\pi}_0[\mathcal{S}_z] = [1/4,\ 0,\ 1/4,\ 0]$. So the vector $\mathbf{m}$ can be computed as:

$$\begin{aligned}\mathbf{m} &= \boldsymbol{\pi}_0[\mathcal{S}_z] \cdot \mathbf{N} \\ &= [1/4 \cdot 1.6461 + 1/4 \cdot 0.4615,\ 1/4 \cdot 0.8615 + 1/4 \cdot 0.6153, \\ &\quad 1/4 \cdot 0.1154 + 1/4 \cdot 1.1538,\ 1/4 \cdot 0.0385 + 1/4 \cdot 0.3846] \\ &= [0.5269,\ 0.3692,\ 0.3173,\ 0.10577] \end{aligned} \tag{3.4}$$

Finally, the measure of mean time to absorption, given $\boldsymbol{\pi}_0 = [1/4,\ 0,\ 1/4,\ 0, 1/2,\ 0]$, is

$$[0.5269 + 0.3692 + 0.3173 + 0.10577] = 1.3192.$$

Since our real interest is in the measure of mean time to absorption, not the fundamental matrix itself, for easier computation, we can avoid direct computation of the inverse that we discussed earlier. To do so, we rewrite the above equation:

$$\begin{aligned}\mathbf{m} &= \boldsymbol{\pi}_0[\mathcal{S}_z] \cdot \mathbf{N} \\ \mathbf{m}(\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]) &= \boldsymbol{\pi}_0[\mathcal{S}_z] \cdot \mathbf{N} \cdot (\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]) \\ \mathbf{m}(\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]) &= \boldsymbol{\pi}_0[\mathcal{S}_z] \\ -\mathbf{m}(\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]) &= -\boldsymbol{\pi}_0[\mathcal{S}_z] \\ \mathbf{m}(\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] - \mathbf{I}) &= -\boldsymbol{\pi}_0[\mathcal{S}_z] \end{aligned} \tag{3.5}$$

From the last step, we can obtain the solution vector $\mathbf{m}$ by solving the above linear system. The benefits of the method are obvious: in terms of storage, we now only need to store the matrix $\mathbf{P} - \mathbf{I}$ which is typically sparse; in terms of computation efficiency, the linear system can be solved in a very similar way as solving for steady-state distribution for an ergodic DTMCs.

Now let's see an example by considering the DTMC as shown in Figure 3.2 again. Let $\mathbf{m} = [\,a,\, b,\, c,\, d\,]$, and multiply out the matrix equation,

$$[\,a,\, b,\, c,\, d\,] \cdot \left[\begin{array}{c|cccc} & a & b & c & d \\ \hline a & -1 & 1/2 & 0 & 0 \\ b & 3/4 & -1 & 1/8 & 0 \\ c & 0 & 1/3 & -1 & 1/3 \\ d & 0 & 0 & 1/5 & -1 \end{array}\right] = [\,-1/4,\, 0,\, -1/4,\, 0\,]$$

we obtain the following system of equations:

$$\begin{aligned} -a + 3/4\, b &= -1/4 \\ 1/2\, a - b + 1/3\, c &= 0 \\ 1/8\, b - c + 1/5\, d &= -1/4 \\ 1/3\, c - d &= 0 \end{aligned}$$

Through a little work by hand, we obtain:

$$\mathbf{m} = [\,137/260,\ 24/65,\ 33/104,\ 11/104\,] \tag{3.6}$$

which exactly matches the result of equation 3.4, which are obtained by computing the inverse directly through a matrix calculator (that's why the numbers appear in floating point format).

Besides the measure of mean time to absorption, another natural question to ask on an absorbing DTMC is that what is the probability that the DTMC eventually reached each of the absorbing states. That is, what is the probability distribution when time goes to infinity. Formally, we want to know the solution for:

$$\lim_{n \to \infty} \Pr\{X(n) = i\}$$

for all absorbing states $i \in \mathcal{S}_a$.

Intuitively, we first try out $\mathbf{P}^n$ and come up with the following generalization:

$$\mathbf{P}^n = \left[\begin{array}{c|c} \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^n & (\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^{n-1} + \cdots + \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] + \mathbf{I}) \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right]$$

From earlier discussions, we know $\lim_{n\to\infty} \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^n = 0$; and

$$\lim_{n\to\infty} (\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]^n + \cdots + \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] + \mathbf{I}) = \mathbf{N}.$$

So we have:

$$\mathbf{P}^\infty = \lim_{n\to\infty} \mathbf{P}^n = \left[\begin{array}{c|c} \mathbf{0} & \mathbf{N} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right] \tag{3.7}$$

From transient analysis, we know

$$\begin{aligned} \boldsymbol{\pi}_\infty &= \lim_{n\to\infty} \boldsymbol{\pi}_n \\ &= \lim_{n\to\infty} \boldsymbol{\pi}_0 \mathbf{P}^n \\ &= \boldsymbol{\pi}_0 \lim_{n\to\infty} \mathbf{P}^n \\ &= [\, \boldsymbol{\pi}_0[\mathcal{S}_z],\ \boldsymbol{\pi}_0[\mathcal{S}_a] \,] \cdot \left[\begin{array}{c|c} \mathbf{0} & \mathbf{N} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right] \\ &= [\, \mathbf{0},\ \boldsymbol{\pi}_0[\mathcal{S}_z] \cdot \mathbf{N} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] + \boldsymbol{\pi}_0[\mathcal{S}_a] \,] \\ &= [\, \mathbf{0},\ \mathbf{m} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] + \boldsymbol{\pi}_0[\mathcal{S}_a] \,] \end{aligned} \tag{3.8}$$

The last equations 3.8 says that the *limiting distribution* of an absorbing DTMC can be determined by the chance of entering each of the absorbing states each time the DTMC visits a transient state, and adding the probability that the process started in each absorbing state.

Continue from the previous DTMC example as shown in Figure 3.2, we have

$$\mathbf{m} = [\, 137/260,\ 24/65,\ 33/104,\ 11/104 \,],$$

$$\boldsymbol{\pi}_0 = [1/4,\ 0,\ 1/4,\ 0, 1/2,\ 0],$$

and

$$\mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] = \begin{bmatrix} 1/2 & 0 \\ 1/8 & 0 \\ 0 & 1/3 \\ 0 & 4/5 \end{bmatrix}$$

By equation 3.8, we have

$$\begin{aligned} \mathbf{P}_\infty =& [\,\mathbf{0} \mid \mathbf{m} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] + \boldsymbol{\pi}_0[\mathcal{S}_a]\,] \\ =& [\,0,\ 0,\ 0,\ 0 \ \mid\ [\,0.30962,\ 0.19038\,] + [\,1/2,\ 0\,]\,] \\ =& [\,0,\ 0,\ 0,\ 0 \ \mid\ 0.80962,\ 0.19038\,] \end{aligned}$$

Therefore, the DTMC is absorbed into state e with probability 0.80962, and into state f with probability 0.19038.

Finally, we note that unlike stationary distributions on ergodic DTMCs, the limiting distribution of absorbing DTMCs depends on the initial distribution.

CHAPTER 4. TRADITIONAL MODEL CHECKING

In the field of formal methods in computer science, *model checking* refers to the process of checking whether a given a model structure meets a specification of a relevant system. The main components of traditional model checking technique include: A logic formalism that describes formal rules governing property specifications; a model structure in the form of a finite state machine (FSM[1]) that represents an abstraction of a real system, and a model checker that will then test if the property is true or false automatically on the model structure.

Typically, the specification requirements concerns about the correctness behavior of a system (hardware or software) including *safety* properties such as "no collision shall occur in a traffic light control system", *liveliness* properties such as "eventually, a service is offered", or *fairness constraints* such as "as long as there is a request, ack message is sent infinitely often", etc.. One commonality of these types of properties is that it does not depend on the time or probability to reach them, but only on which states can be reached. For such properties, traditional logic formalisms such as linear-time temporal logic (LTL) [58], computation tree logic (CTL) [20], and computation tree logic star (CTL*) [31] are powerful, widely-used logics [2]. These logics use atomic propositions and boolean operators to build up complicated expressions describing state transition properties.

For this chapter in the following, we first describe a classic model structure that is commonly used in traditional model checking, and then describe the most well known temporal logics namely LTL, CTL, and CTL*.

[1] An FSM is a directed graph similar to DTMC but transitions between states are based on non-deterministic choice rather than probability.

[2] As a historical note, LTL was invented first by philosopher Arthur Prior in the 1960s; and has been further developed on by computer scientists A. Pnueli, Z. Manna, and other logicians; CTL was invented by E. Clarke and E. A. Emerson in the early 1980s; and CTL* was introduced in 1986 by E. A. Emerson and J. Halpern [37].

4.1 Kripke Structure

Definition 21 (Kripke structure)**.** *By [37], a Kripke structure K over a set of atomic proposition $\mathcal{AP}$ is a four tuple $K = (\mathcal{S}, \mathcal{S}_0, T, L)$ where*

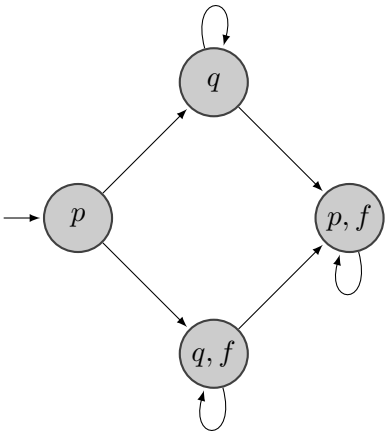
- *$\mathcal{S}$ is a finite set of states.*
- *$\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial states.*
- *$T \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation such that every $s \in \mathcal{S}$ has some $s' \in \mathcal{S}$ with $(s, s') \in T$.*
- *$L : \mathcal{S} \to 2^{\mathcal{AP}}$ is a labeling function that maps each state to a set of atomic propositions true in that state.*

A *path* in a Kripke structure K is an infinite sequence of states, denoted by $\pi = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in T$, for all $i : i \geq 0$; we use π^i to denote the suffix of π starting at s_i (i.e., π^i is an infinite sequence of states, whereas π_i denotes a state s_i). A *state, path formula* is one that either holds, or fails to hold, for a particular state, path, respectively. For a given structure K, if f is a *state formula* (of a certain logic) and f holds *true* at state s in K, then it is written by $s \models f$; otherwise, it is written by $s \not\models f$ (where $\models$ denotes a "satisfaction" relation between a state and a temporal logic formula, and $s \models f$ is read as s satisfies f). Similarly, if f is a *path formula* (of a certain logic) and f holds *true* along path π in K, then it is written by $\pi \models f$; otherwise it is written by $\pi \not\models f$. Figure 4.1(a) shows an example of Kripke structure, with initial state p. [20, 37] discusses more in detail about the model.

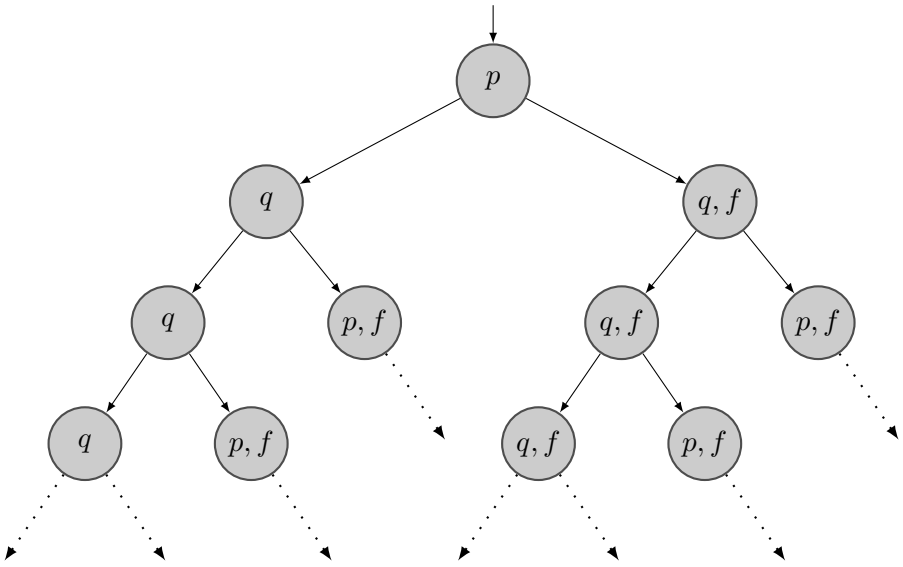
4.2 Computation Tree Logic (CTL)

Computation tree logic [19], or CTL for short, is a formal logic that is designed to express properties related to the correctness of system behaviors. The interpretation of the logic is described using a computation tree. Figure 4.1(b) shows an example of infinite computation tree obtained by unwinding the Kripke structure of figure 4.1(a).

Syntactically, a CTL formula has two *path quantifiers*: A and E which denote "all" computation paths and "there exists" a computation path, respectively. A CTL path formula must have one of the following temporal path operators: *next* operator X, *until* operator U, *weak until* operator V, *all*



(a) A Kripke Structure Example



(b) An Infinite Computation Tree

Figure 4.1: Unwinding the Kripke Structure $\rightarrow$ Obtaining a Computation Tree

operator G, and *future* operator F. Among those, X, G, and F are unary operators that take one state formula as an argument, and for all other path operators, namely U and V, are binary operators that take two state formula as arguments.

Informally, let $f,\ f_1,\ f_2$ be state formulas, given a path π with starting state π_0, then, Xf means f holds next time at π_1; Ff means f holds sometime in the future along π; Gf means f holds globally in the future; $f_1 U f_2$ means f_2 holds at some state $\pi_j, j \geq 0$, and before that, f_1 holds at all states π_i, with $0 \leq i < j$; and $f_1 V f_2$ means either $f_1 U f_2$ holds along π, or f_1 holds at all states along the path. The following gives the formal syntax and semantics of CTL.

Definition 22 (CTL Syntax)**.** *Let $\mathcal{AP}$ be the set of atomic propositions, and $p \in \mathcal{AP}$, then CTL syntax can be recursively defined as follows:*

$$\phi \ ::= \ true \mid false \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid A\psi \mid E\psi$$

where ψ is a path formula and is recursively defined as follows:

$$\psi \ ::= \phi \, U \, \phi \mid \phi \, V \, \phi \mid G\phi \mid F\phi \mid X\phi.$$

Definition 23 (CTL Semantics)**.** *The semantics of CTL formulas can be recursively defined as follows:*

- $s \models true$ *and* $s \not\models false$ *for all* $s \in \mathcal{S}$.
- $s \models p$ *iff* $p \in L(s)$.
- $s \models \neg\phi$ *iff* $s \not\models \phi$.
- $s \models \phi_1 \wedge \phi_2$ *iff* $s \models \phi_1$ *and* $s \models \phi_2$.
- $s \models \phi_1 \vee \phi_2$ *iff* $s \models \phi_1$ *or* $s \models \phi_2$.
- $s \models A\psi$ *iff for every path with* $\pi_0 = s$, $\pi \models \psi$.
- $s \models E\psi$ *iff there exists a path with* $\pi_0 = s$, $\pi \models \psi$.
- $\pi \models X\phi$ *iff* $\pi_1 \models \phi$.

- $\pi \models \phi_1 \, U \, \phi_2$ *iff* $\exists j \geq 0, \pi_j \models \phi_2$ *and* $\forall i : 0 \leq i < j, \pi_i \models \phi_1$.
- $\pi \models \phi_1 \, V \, \phi_2$ *iff either* $\exists j \geq 0, \pi_j \models \phi_2$ *and* $\forall i : 0 \leq i < j, \pi_i \models \phi_1$, *or* $\forall i : 0 \leq i, \pi_i \models \phi_1$.

Also, we have:

$$AF\phi \equiv A[true \, U \, \phi]$$

$$EF\phi \equiv E[true \, U \, \phi]$$

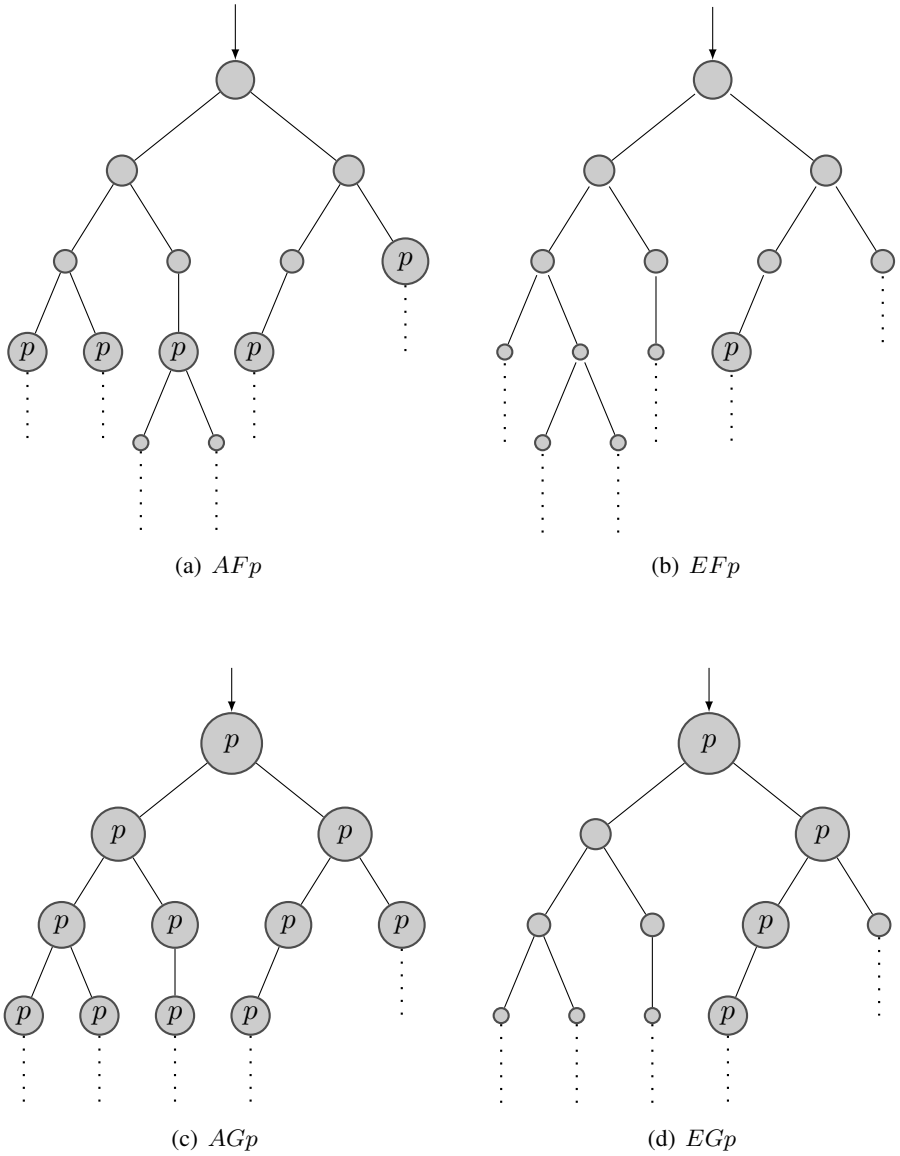
$$\neg AF\phi \equiv \neg A[true \, U \, \phi] \equiv EG\neg\phi$$

$$\neg EF\phi \equiv \neg E[true \, U \, \phi] \equiv AG\neg\phi$$

Since the top-level CTL formula ϕ is a state formula, the path quantifier must be paired with a path formula to be valid. By using concrete path formulas, we get ten pairs of state formulas in the form of $AX, EX, AF, EF, AG, EG, AU, EU, AV, EV$, and each of these state formulas can be expressed in terms of the three operators EX, EG, and EU [18, 37]:

$$\begin{aligned}
AX \; f &= \neg EX(\neg f) \\
EF \; f &= E[true \; U \; f] \\
AG \; f &= \neg EF(\neg f) \\
AF \; f &= \neg EG(\neg f) \\
A[f_1 \; U \; f_2] &\equiv \neg(E[\neg f_2 \, U(\neg f_1 \wedge \neg f_2)] \vee EG\neg f_2) \\
&\equiv \neg E[\neg f_2 \, U(\neg f_1 \wedge \neg f_2)] \wedge \neg EG\neg f_2 \\
A[f_1 \; V \; f_2] &\equiv \neg E[\neg f_2 \, U(\neg f_1 \wedge \neg f_2)] \\
E[f_1 \; V \; f_2] &\equiv \neg A[\neg f_2 \, U(\neg f_1 \wedge \neg f_2)]
\end{aligned}$$

Figure 4.2 shows example structures that satisfy AFp, EFp, AGp, EGp, respectively. For the structure of AFp, along each path, an atomic proposition p is labeled, which implies property p holds true in that state. For the structure of EFp, there is one path where p is labeled, so EFp holds true with respect to the initial state. For the structure of AGp, p is labeled along every path and every state, hence AGp



(a) AFp (b) EFp

(c) AGp (d) EGp

Figure 4.2: Example Structures Whose Starting States Satisfy the CTL Formulas, Respectively.

holds true with respect to the initial state. For the structure of EGp, there exists one path along which p is labeled on every state, hence EGp holds true with respect to the initial state of the given structure.

Finally, given a Kripke structure K, and a CTL state formula ϕ, the *satisfaction set*, denoted by $Sat(\phi)$, is defined by:

$$Sat(\phi) = \{s \in \mathcal{S} \mid s \models \phi\}$$

and K satisfies ϕ if and only if ϕ holds in all its initial states s_0. Interested readers should refer to [37] for the detailed discussion on computing the set $Sat(\phi)$.

4.3 Linear-time Temporal Logic (LTL)

Linear-time temporal logic [58], or LTL for short, is designed to express properties related to *linear time* paths. Like CTL, LTL has the same set of temporal path operators, namely X, U, V, F, and G, and they have the exact same interpretations on paths as CTL. However, unlike CTL, LTL's top level formulas are path based, rather than state-based. The following gives formal definitions about its syntax and semantics.

Definition 24 (LTL Syntax)**.** *Let $\mathcal{AP}$ be a set of atomic propositions, and $p \in \mathcal{AP}$, then LTL syntax can be recursively defined as follows:*

$$\psi ::= p \mid (\neg\psi) \mid (\psi \wedge \psi) \mid G\psi \mid F\psi \mid X\psi \mid \psi\ U\ \psi \mid \psi\ V\ \psi.$$

where X, U, V G, and F are temporal operators that denote "next", "until", "weak until (a variant of until)", "all (states)", and "there exists (a future state)", respectively, for a path.

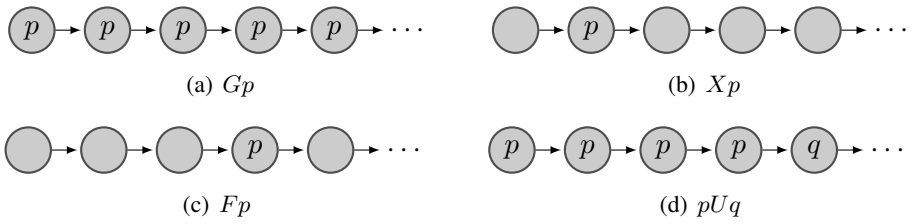
Definition 25 (LTL Semantics)**.** *The semantics of LTL formulas can be recursively defined as follows:*

- $\pi \models p$ *iff* $p \in L(\pi_0)$.
- $\pi \models \neg\psi$ *iff* $\pi \not\models \psi$.
- $\pi \models \psi_1 \wedge \psi_2$ *iff* $\pi \models \psi_1$ *and* $\pi \models \psi_2$.
- $\pi \models X\psi$ *iff* $\pi^1 \models \psi$.

- $\pi \models G\psi$ *iff* $\forall i \geq 0, \pi^i \models \psi$.
- $\pi \models F\psi$ *iff* $\exists i \geq 0, \pi^i \models \psi$.
- $\pi \models \psi_1 \, U \, \psi_2$ *iff* $\exists j \geq 0, \pi^j \models \psi_2$ *and* $\forall i : 0 \leq i < j, \, \pi^i \models \psi_1$.
- $\pi \models \psi_1 \, V \, \psi_2$ *iff either* $\exists j \geq 0, \pi^j \models \psi_2$ *and* $\forall i : 0 \leq i < j, \pi^i \models \psi_1$, *or* $\forall i : 0 \leq i, \pi^i \models \psi_1$.

Figure 4.3 illustrates example paths corresponding to the semantics of LTL formulas, respectively. Specifically, Figure 4.3(a) shows p is labeled everywhere along the path, so Gp holds for the given path. Figure 4.3(b) shows p is labeled in the next state relative to the beginning of the path, so Xp holds for the given path. Figure 4.3(c) shows p is labeled somewhere in a state along the path, so Fp holds for the given path. Figure 4.3(d) shows q is labeled somewhere in a state along the path, before that, p is labeled in all states along the path, so pUq holds for the given path.

Note that although the syntax of LTL does not have path quantifier A or E as CTL does, at the outermost level, LTL paths are viewed as *all* paths [37]. For example, an LTL formula Gp is the same as saying AGp, FGp is the same as saying $A(FGp)$, $Xp \vee XXp$ is the same as saying $A(Xp \vee XXp)$, and so on.

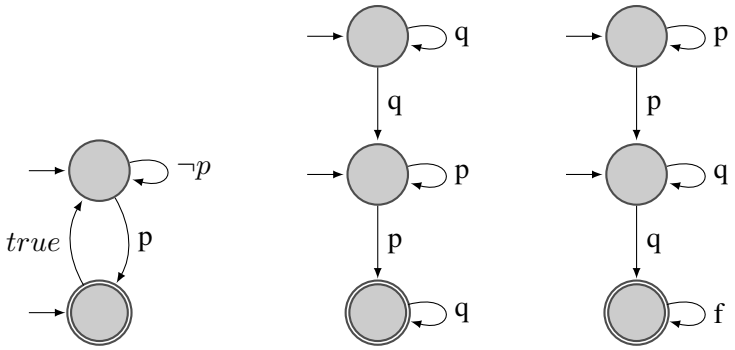


(a) Gp (b) Xp (c) Fp (d) pUq

Figure 4.3: LTL Formula Examples.

Given a Kripke structure K, an LTL formula expresses a set of paths in K that are satisfied by the formula. Figure 4.4 illustrates some examples of LTL formulas in B$\ddot{u}$chi automata representation, where nodes of double circles represent the accepting states. B$\ddot{u}$chi automata is an important notion in model checking [64]. Typically, given a model structure K and an LTL formula ψ, a B$\ddot{u}$chi automaton is built for $\neg\psi$. By computing a cross product of the automata and K, a set of paths that are both in K and accepted by the automata can be produced; if the set is empty, then K satisfies ψ, otherwise the

output of the paths are regarded as a counter example. In this discussion, we omit the details of LTL model checking and refer interested readers to [64, 65] for the topic of applying Büchi's techniques to LTL, and [40] for direct property specification in automata.



(a) GFp (b) Fp (c) pUq

Figure 4.4: Example LTL formulas in Büchi Automata Representation.

4.4 LTL vs. CTL

From previous discussion, we know there is a lot in common between LTL and CTL formulas. In this section, we illustrate two different scenarios where LTL and CTL can have different expressive powers.

From Figure 4.5, we see for every path with respect to the initial state of the Kripke structure K, there exists a future state i such that p is labeled in every state along the future path π^i. For example, we have paths:

$$p,\ p,\ p,\ p,\ \cdots,$$

$$p,\ \neg p,\ p,\ p,\ p,\ \cdots,$$

$$p,\ p,\ p,\ \neg p,\ p,\ p,\ p,\ \cdots,$$

etc.. So it satisfies LTL formula FGp. However, this is not true with CTL formula $AFAGp$, because there exists a path $p, p, p, \cdots$, none of the states from which it satisfies AGp, since from any state of the path, there exists a path where $\neg p$ holds, which contradicts AGp.

Now, let's look at Figure 4.6. There is a path $p, \neg p, p, \neg p, \cdots$ where p and $\neg p$ are labeled alternatively along the path, which make the LTL formula FGp not satisfiable, because this formula requires to have labeled p everywhere eventually along every path. In contrast, this structure satisfies CTL formula $AFEGp$, since along all paths, there exists a future state from which there exists a path where p holds along the path.

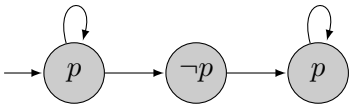


Figure 4.5: $K \models_{LTL} FGp,\ K \not\models_{CTL} AFAGp$

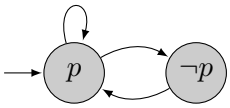


Figure 4.6: $K \not\models_{LTL} FGp,\ K \models_{CTL} AFEGp$

4.5 Computation Tree Logic Star (CTL*)

CTL* [31] is an extension of CTL that unifies LTL and CTL. That is, both LTL and CTL are subsets of CTL*. Its complete syntax consists of state formulas

$$\phi \ ::= \ true \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid A\psi \mid E\psi,$$

and path formulas

$$\psi \ ::= \phi \mid (\neg\psi) \mid (\psi \wedge \psi) \mid (\psi \vee \psi) \mid \psi \ U \ \psi \mid \psi \ V \ \psi \mid G\psi \mid F\psi \mid X\psi.$$

Unlike LTL or CTL, with CTL*, ψ and ϕ are mutually recursive. The semantics of CTL* is omitted here, because they are merely the combination of CTL and LTL. As such, CTL* is strictly more expressive than both CTL and LTL. For example, the CTL* formula:

$$AFGp \ \vee \ AFEGp$$

can neither be expressed by CTL, nor LTL. As another example, $E(GFp)$ means there is a path along which p holds true infinitely often.

Clearly, CTL is a restricted subset of CTL* that permits only branching-time operators: each of the linear-time operators G, F, X, U, and V must be immediately preceded by a path quantifier, e.g. $AG(EF(p))$. LTL is also a restricted subset of CTL* that consists of formulas that have the form Af where f is a path formula in which the only state subformulas permitted are atomic propositions. e.g. $A(FGp)$.

CHAPTER 5. PROBABILISTIC MODEL CHECKING

From the previous chapter, we learned that temporal logics such as LTL [58] and CTL [19] are powerful, widely used logics for properties that depend only on which states can be reached, such as "the system never reaches a deadlocked state", and not on the time or probability to reach them. To express performance related properties, these logics have been extended by adding or modifying operators (e.g. PLTL [22], PCTL [33], pCTL* [3], DCTL[25], CSL [6], asCSL [5], CSL$^{\mathrm{TA}}$ [28]), and can express properties such as "with probability at least 0.98, the system will not reach a deadlocked state before time 50."

For probabilistic model checking, the underlying model structure is typically a Markov chain, either discrete time or continuous time, which have been discussed in an earlier chapter. So for this chapter, we describe several stochastic (a.k.a. probabilistic) logics that are most relevant to our work.

5.1 Probabilistic Computation Tree Logic (PCTL)

Probabilistic Computation Tree Logic [33], or PCTL for short, is an extension of CTL for expressing real-time and probabilities in systems. From the perspective of language specification, the main difference between PCTL and CTL is that, PCTL allows specification of probability bounds on paths, with or without deadline t. As such, PCTL replaces the CTL path quantifiers of A and E with a universal $P_{\bowtie v}$ operator, which says that the probability of all paths satisfying ϕ is $\bowtie$ v, where $\bowtie$ is a comparison operator, P means "probability", v is a probability value specified by user, and ϕ is a state formula similar to that of CTL.

Definition 26 (PCTL syntax)**.** *Let $\mathcal{AP}$ be a set of atomic propositions, and $p \in \mathcal{AP}$, the syntax of PCTL can be defined recursively as follows:*

$$\phi \ ::= \ p \mid \neg\phi \mid (\phi \vee \phi) \mid P_{\bowtie v}\psi, \qquad \psi \ ::= \ \phi \, \mathrm{U}^{\leq t} \, \phi \mid X\phi$$

where $t \in \mathbb{N} \cup \{\infty\}$, $\bowtie \in \{\geq, \leq, >, <\}$, *and* $v \in [0, 1]$.

Like CTL, the top level formula ϕ is a state formula in PCTL. Unlike CTL, however, PCTL is interpreted over a variant of DTMC structure. The following gives formal definition of PCTL's interpretation structure.

Definition 27 (Interpretation Structure of PCTL)**.** *Let* $D = (\mathcal{S}, \boldsymbol{\pi}_0, \mathbf{P})$ *be a DTMC, and* $L : \mathcal{S} \to 2^{\mathcal{AP}}$ *be a labeling function, where* $\mathcal{AP}$ *denotes a set of atomic propositions, then an underlying structure of PCTL is a pair* (D, L).

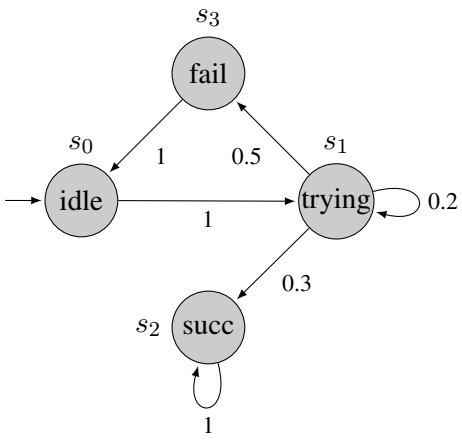


Figure 5.1: An Example of Discrete Time Probabilistic Structure

Figure 5.1 shows an example PCTL interpretation structure, with $\mathcal{S} = \{s_0, s_1, s_2, s_3\}, \boldsymbol{\pi}_0[s_0] = 1, \boldsymbol{\pi}_0[s_1] = \boldsymbol{\pi}_0[s_2] = \boldsymbol{\pi}_0[s_3] = 0, \mathcal{AP} = \{fail, idle, trying, succ\}$, and

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.2 & 0.3 & 0.5 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Definition 28 (PCTL Path [33]). *A path π in a probabilistic structure is an infinite sequence of states $\pi = s_0, s_1, s_2, \cdots$ such that for each $i > 0, i \in \mathbb{N}, \mathbf{P}[s_i, s_{i+1}] > 0$.*

Definition 29 (Prefix). *A prefix in $D = (\mathcal{S}, s_0, \mathbf{P}, L)$ is a finite sequence $\mathbf{p} = (\pi_0, \pi_1, \cdots, \pi_{n-1}) \in \mathcal{S}^n$, or an infinite sequence $\mathbf{p} = (\pi_0, \pi_1, \cdots) \in \mathcal{S}^\omega$, where $|\mathbf{p}| = n \in \mathbb{N} \cup \{\omega\}$ is the length of the sequence.*

Definition 30 (PCTL Semantics). *Let $p, q \in \mathcal{AP}$, and ϕ be a state formula, the semantics of PCTL can be defined recursively as follows:*

- $s \models p$ *iff* $p \in L(s)$.
- $s \models \neg\phi$ *iff* $s \not\models \phi$.
- $s \models \phi_1 \vee \phi_2$ *iff* $s \models \phi_1$ *or* $s \models \phi_2$.
- $\pi \models \phi_1 \, U^{\leq t} \, \phi_2$ *iff* $\exists j \leq t : \pi_j \models \phi_2$ *and* $\forall i : 0 \leq i < j, \, \pi_i \models \phi_1$.
- $\pi \models X\phi$ *iff* $\pi_1 \models \phi$.
- $s \models P_{\bowtie v}\psi$ *iff the probability measure, with respect to the initial state s, of the set of paths π for which $\pi \models \psi$ is $\bowtie v$.*

Note that the shorthand $P_{\bowtie v}\psi$ concerns the quantity of probability measure, which we have discussed before in general cases. In this context, the probability space is a pair of $(\mathcal{X}, \Sigma)$, where $\mathcal{X}$ is the set of paths starting in s and Σ is a σ-algebra over $\mathcal{X}$ generated by sets of paths with a common finite prefix $\mathbf{p}$. The probability measure, with respect to starting state $s = \pi_0$, for the set of paths that satisfy the path formula ψ is then defined in terms of the prefix as follows:

$$\Pr(\{\pi \mid \mathbf{p} = \pi_0, \cdots, \pi_n\}) = \mathbf{P}[\pi_0, \pi_1] * \cdots * \mathbf{P}[\pi_{n-1}, \pi_n],$$

for $n > 0$, and $\Pr(\{\pi \mid \mathbf{p} = \pi_0\}) = 1$, for $n = 0$. Then, the probability measure, with respect to the starting state s, of the set of paths π for which $\pi \models \psi$ is defined as $\Pr(\{\pi \mid \pi \in \mathcal{X}, \pi \models \psi, s = \pi_0\})$. For more details regarding PCTL's probability measure, interested reader should refer to [33].

Take Figure 5.1 as an example, one can have PCTL specifications such as

- $P_{\geq 0.95} true\, U^{\leq \infty} succ$ asserts that with at least 0.95 probability that the system will eventually reach the $succ$ state.
- $P_{\geq 0.5} true\, U^{\leq 100}\, fail$ asserts that with at least 0.5 probability that the system will reach $fail$ state before 100 time unit.

Like CTL, PCTL allows nesting of P, X, U operators in the pairs of PX and PU.

- $A(\psi) \equiv P_{\geq 1}(\psi)$
- $E(\psi) \equiv P_{>0}(\psi)$

5.1.1 Algorithm For The Case of $P_{\bowtie v} X\phi$

In this case, we first compute probability measure for the formula $P\bowtie vX\phi$, then compare the quantity to v, for each starting state $s \in \mathcal{S}$. Let $\mathbf{f}$ be a vector such that $\mathbf{f}[j] = 1$ if and only if $j \models \phi$ for all $j \in \mathcal{S}$, then following the semantic of $X\phi$ and the definition of its probability measure, we have

$$\begin{aligned} \boldsymbol{\pi}[i] &= \sum_{\forall j} \mathbf{P}[i,j] \cdot \mathbf{f}[j] = \mathbf{P}[i, \mathcal{S}] \cdot \mathbf{f} \\ \boldsymbol{\pi} &= \mathbf{P} \cdot \mathbf{f} \end{aligned} \tag{5.1}$$

Then, a state $i \models P_{\bowtie v}(X\phi)$ if and only if $\boldsymbol{\pi}[i] \bowtie v$. For convenience, let $\mathbf{h}$ be the final solution vector such that $\mathbf{h}[i] = 1$ if and only if $i \models P_{\bowtie v}(X\phi)$, then we have:

$$\mathbf{h}[i] = 1 \ \text{ iff } \ \boldsymbol{\pi}[i] \bowtie v \tag{5.2}$$

Consider Figure 5.1 for example, suppose we want to know which states satisfy $P_{<3/4}(X\neg fail)$. First, we have $\mathbf{f} = [1, 1, 1, 0]$, corresponding to states $idle$, $trying$, $succ$, and $fail$, respectively. Then,

we have

$$\pi = \mathbf{P} \cdot \mathbf{f} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.2 & 0.3 & 0.5 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \\ 1 \\ 1 \end{bmatrix}$$

Therefore, the solution vector is $\mathbf{h} = [0, 1, 0, 0]$, i.e., only state *trying* satisfy the formula $P_{<3/4}(X \neg fail)$.

5.1.2 Algorithm For The Case of $P_{\bowtie v}\phi_1 \, U^{\leq t}\phi_2, \;\; t \in \mathbb{N} \cup \{\infty\}$

In this case, the computation involves two main steps: the first main step is to do a simple model modification. The second main step is to compute the probability measure based on the modified model.

Given the semantics of the path formula $\phi_1 \, U^{\leq t}\phi_2$, we can modify the model such that the states satisfying ϕ_2 are made absorbing (called *success* states); then the states that neither satisfy ϕ_1 nor ϕ_2 are made absorbing (called *failure* states), and the rest of states are unchanged. This makes sense, because by semantics once a DTMC reaches a success state then computation of the relevant path stops at that state; similarly, once if a DTMC reaches a failure state then the probability of the relevant path must be zero and any outgoing edge from there is not needed. For the following discussion, let $\mathcal{S}_a \subseteq \mathcal{S}$ to denote the set of absorbing states, and $\mathcal{S}_z \subseteq \mathcal{S}$ to denote the set of transient states.

Now, let $P(t, s)$ denote the probability measure for the set of paths that satisfy $\phi_1 \, U^{\leq t}\phi_2$, with respect to starting state s; let t be a nonnegative integer, then $P(t, s)$ can be computed by the following recursive algorithm.

Algorithm 5.1 Compute $P(t, s)$

if $s \models \phi_2$ **then**
  return 1
end if
if $s \not\models \phi_1$ or $t = 0$ **then**
  return 0
end if
return $\Sigma_{s' \in \mathcal{S}} \mathbf{P}(s, s') \cdot P(t - 1, s')$

The above algorithm terminates when it encounters one of the following three base cases:

- . If s is a target state that satisfies ϕ_2, then regardless of time t, the probability measure is 1 and the program terminates.
- Otherwise, if s is a *failure* state that does not satisfy ϕ_1, then regardless of time t, the probability measure is 0 and the program terminates;
- Otherwise, if time t reaches 0, then the probability measure is 0 and the program terminates.

The only remaining case is that t is greater than 0, and s satisfies ϕ_1, in this case, it recursively computes its probability measure until one of the above stopping cases is met. The detailed proof about the correctness of the algorithm is presented in [33].

Alternatively, the formula can be computed by using matrix-vector product, which yields a vector of probability measures with respect to each starting state $s \in \mathcal{S}$. Specifically, let $\mathbf{P}$ be the probability transition matrix corresponding to the new DTMC, let $\mathbf{f}$ be the vector such that $\mathbf{f}[i] = 1$ if $i \models \phi_2$ and 0 otherwise. Then, for a starting state i, we can determine the probability measure by

$$\Pr\{X(t) \models \phi_2 \mid X(0) = i\} \tag{5.3}$$

which can be rewritten as

$$\sum_{\forall j} \Pr\{X(t) = j \mid X(0) = i\} \cdot \mathbf{f}[j] \tag{5.4}$$

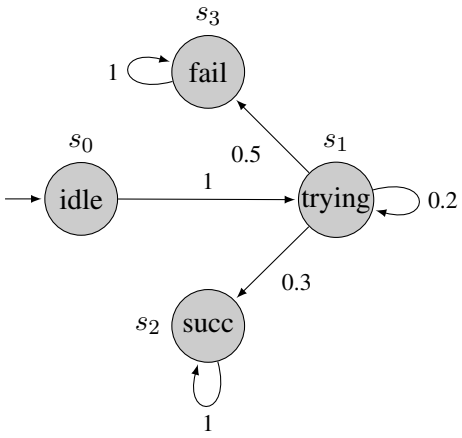
Then, for all starting states $i \in \mathcal{S}$, we have

$$\boldsymbol{\pi}_t = \mathbf{P}^t \cdot \mathbf{f} = \mathbf{P} \cdot \boldsymbol{\pi}_{t-1}, \text{ with } \boldsymbol{\pi}_0 = \mathbf{f}. \tag{5.5}$$

Finally, we have $\mathbf{h}[i] = 1$ iff $\boldsymbol{\pi}_t[i] \bowtie v$.

Consider Figure 5.1 for example, suppose we want to know which states satisfy $P_{\geq 0.3} \neg \mathit{fail}\, U^{\leq 3} \mathit{succ}$. First, we observe state succ (success state) is already absorbing, so no need to change; we make state

fail (failure state) absorbing; states *idle* and *trying* are unchanged. Figure 5.2(a) and 5.2(b) show the modified DTMC model and the corresponding probability transition matrix, respectively.



(a) Modified Example Structure of Figure 5.1

$$\mathbf{P} = \begin{bmatrix} & idle & trying & succ & fail \\ idle & 0 & 1 & 0 & 0 \\ trying & 0 & 0.2 & 0.3 & 0.5 \\ succ & 0 & 0 & 1 & 0 \\ fail & 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) Probability Transition Matrix

Figure 5.2: Modification Example

Next, we compute the probability measure like the transient analysis for DTMC.

$$\boldsymbol{\pi}_0 = \mathbf{f} = [0, 0, 1, 0]$$

$$\boldsymbol{\pi}_1 = \mathbf{P} \cdot \boldsymbol{\pi}_0 = [0, 0.3, 1, 0]$$

$$\boldsymbol{\pi}_2 = \mathbf{P} \cdot \boldsymbol{\pi}_1 = [0.3, 0.36, 1, 0]$$

$$\boldsymbol{\pi}_3 = \mathbf{P} \cdot \boldsymbol{\pi}_2 = [0.36, 0.372, 1, 0]$$

Finally, we have $\mathbf{h} = [1, 1, 1, 0]$, i.e., the *idle*, *trying*, and *succ* states satisfy $P_{\geq 0.3}\neg\text{fail}\, U^{\leq 3}\text{succ}$.

While the above algorithms are good when t is bounded, it requires infinite computation when t is unbounded [33]. We now discuss a computation method when t goes to infinity. For simplicity, let $P_{\bowtie v}\phi_1\, U^{\leq\infty}\phi_2 \equiv P_{\bowtie v}\phi_1\, U\phi_2$. To compute a state i that satisfy $P_{\bowtie v}\phi_1 \mathrm{U} \phi_2$, first we do the same DTMC modification as we discussed for the time bounded version. In the new DTMC, we want to measure

$$\boldsymbol{\pi}_\infty[i] = \lim_{t\to\infty} \mathbf{P}^t \cdot \mathbf{f}$$

for all state $i \in \mathcal{S}$. By the semantics of the formula, we know the set of failure states always have

probability 0, and the set of success states always have probability 1. So it remains to compute the set of transient states; note that for those which are not transient states, that means they are not able to reach the success states therefore they are counted as failure states as well. As per the discussion earlier in section 3.2.3, for the absorbing part of states, we have

$$\boldsymbol{\pi}_{\infty}[\mathcal{S}_a] = \mathbf{f}[\mathcal{S}_a]$$

while the transient part of states is

$$\boldsymbol{\pi}_{\infty}[\mathcal{S}_z] = \mathbf{N} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \cdot \mathbf{f}[\mathcal{S}_a]$$

which can be written as

$$\begin{aligned} \mathbf{N}^{-1}\boldsymbol{\pi}_{\infty}[\mathcal{S}_z] &= \mathbf{N}^{-1} \cdot \mathbf{N} \cdot \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \cdot \mathbf{f}[\mathcal{S}_a] \\ (\mathbf{I} - \mathbf{P}[\mathcal{S}_z, \mathcal{S}_z]) \cdot \boldsymbol{\pi}_{\infty}[\mathcal{S}_z] &= \mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \cdot \mathbf{f}[\mathcal{S}_a] \\ (\mathbf{P}[\mathcal{S}_z, \mathcal{S}_z] - \mathbf{I}) \cdot \boldsymbol{\pi}_{\infty}[\mathcal{S}_z] &= -\mathbf{P}[\mathcal{S}_z, \mathcal{S}_a] \cdot \mathbf{f}[\mathcal{S}_a] \end{aligned} \tag{5.6}$$

From previous chapters, we know $\boldsymbol{\pi}_{\infty}[\mathcal{S}_z]$ can be determined by solving the linear system of equations as shown in 5.6. Finally, we can obtain the whole vector $\mathbf{h}$ by examining $\boldsymbol{\pi}_{\infty}$ in exactly the same way as for time bounded until.

5.2 PLTL and pCTL*

Analogous to PCTL, Courcoubetis et al. [22] describe a probabilistic model checking algorithm for LTL, known as PLTL, and denoted by $P_{\bowtie v}\psi$, where ψ is an LTL path formula, and $P_{\bowtie v}\psi$ means the probability measure of the set of paths π for which $\pi \models \psi$ is $\bowtie v$. From the perspective of language semantics, the difference between PCTL and PLTL is the same as the difference between CTL and LTL. In the former case, the path formula ψ is limited to CTL's path formula which is a subset of LTL's path formula, whereas in the latter case, the path formula ψ is LTL's path formula. Note that the probability operator $P_{\bowtie}$ of PLTL cannot be nested.

pCTL* [3] is a branching time logic similar to PCTL (both are state-based logics). Analogous to CTL*, it is a unification of PCTL and PLTL. In the following, let ϕ denote a state formula and ψ denote a path formula. Its syntax is given by:

$$\phi \ ::= \ p \mid \neg\phi \mid (\phi \vee \phi) \mid P_{\bowtie v}\psi$$

where the path formula ψ is given by

$$\psi \ ::= \phi \mid (\neg\psi) \mid (\psi \vee \psi) \mid \psi \, U \, \psi \mid X\psi.$$

Unlike PCTL, a state formula ϕ is also a path formula in pCTL*, and unlike PLTL, every top-level formula is a state formula in pCTL*. Since the semantics is exactly the same as that of PCTL and PLTL, respectively, we omit the interpretation of pCTL* formulas.

CHAPTER 6. REAL-VALUED PERFORMANCE MODELING FORMALISMS

While probabilistic model checking logics takes a big step towards combining performance analysis and model checking techniques, and can express performance related properties such as "with probability at least 0.95, the system will not reach a deadlocked state before time 100", they are limited to producing only true or false responses, precisely because they are logics (although in practice, a real valued response can sometimes be obtained for the outer most path quantifier).

In this chapter, we introduce a real valued formal language [52], towards the unification of model checking and performance evaluation. The formal language serves a main contribution for this thesis. It is a further generalization of probabilistic model checking approach, in the sense that it expands quantitative analysis from the value range of $\{0, 1\}$ to $[0, \infty)$. The following discusses more details about the language and computation algorithms as well.

6.1 Computation Tree Measurement Language

Computation Tree Measurement Language, or (CTML) for short, is an extension of PCTL. Syntactically, it is similar to CTL and PCTL in a sense that it is a state-based tree structured language. And they share path operators such as *next* (X), *until* (U), and *weak until* (V), etc. Semantically, however, CTML quite differs from PCTL, and perhaps all the formal stochastic logics to our best knowledge thus far, because CTML takes (non-negative) real values as input and output real values, as opposed to true/false (or $1/0$) values as input and output true/false (or $1/0$) values. Although in recent research, some probabilistic model checking tools such as PRISM [42, 44] are able to incorporate operators like $P_?$ (adjusted from $P_{\bowtie}v$ in PCTL) to get probabilistic real values, they cannot "nest" real values, because a stochastic logic, by nature and as a whole, is still a logic.

The real-valued nature of the CTML language is natural for expressing performance questions such as, "what is the probability, the system will reach a deadlock state before time t?". But its significance comes from the nesting power. Through nesting of the real valued formula, not only this departs it from other formal stochastic logics, but also it yields a wider variety of performance queries such as: "when a message is sent, what is the expected time before it is received?", etc.. To our best knowledge, such queries are not expressible in existing formal stochastic logics in the general case.

6.1.1 Basic Definitions

Definition 31 (path)**.** *Let $D = (\mathcal{S}, \mathbf{P}, \boldsymbol{\pi}_0)$ be a CTML interpretation structure, a* path *in D, denoted by π, is defined as an infinite sequence $(s_0, s_1, s_2, \ldots) \in \mathcal{S}^\omega$ such that $\mathbf{P}[s_i, s_{i+1}] > 0, i \geq 0, i \in \mathbb{N}$.*

Definition 32 (state formula)**.** *A CTML state formula ϕ is defined as a function that maps a given state space $\mathcal{S}$ to nonnegative real values:*

$$\phi : \mathcal{S} \to \mathbb{R}^*.$$

Definition 33 (restricted state formula)**.** *A restricted CTML state formula φ is defined as a function that maps a given state space $\mathcal{S}$ to the interval $[0, 1]$:*

$$\varphi : \mathcal{S} \to [0, 1].$$

Definition 34 (path formula)**.** *A CTML* path formula *ψ is a function $\psi : \mathcal{S}^\omega \to \mathbb{R}^*$.*

Definition 35 (restricted path formula)**.** *A CTML* restricted path formula *ϱ is defined as a function that maps from the set of paths to the interval $[0, 1]$.*

$$\varrho : \mathcal{S}^\omega \to [0, 1].$$

For this work, we wish to determine the *expected value* of a path formula ψ. So below, we describe some properties on ψ that will allow us to define a measure for this expected value.

For a given prefix $\mathbf{p} = (\pi_0, \ldots, \pi_{n-1})$ (see definition 29), define $\mathcal{S}^\omega_{\mathbf{p}}$ as the set of all infinite-length paths that start with prefix $\mathbf{p}$. Note that, if the length of $\mathbf{p}$ is $n \in \mathbb{N}$, we have

$$\mathcal{S}^\omega_{\mathbf{p}} = \{\pi_0\} \times \cdots \times \{\pi_{n-1}\} \times \mathcal{S}^\omega; \tag{6.1}$$

otherwise, if the length of $\mathbf{p}$ is ω, then we have $\mathcal{S}^\omega_{\mathbf{p}} = \{\mathbf{p}\}$. Note $\mathcal{S}^\omega_{()} = \mathcal{S}^\omega$, where $()$ denotes the zero-length sequence. Let Σ_s denote the σ-algebra generated by the set $\{\mathcal{S}^\omega_{\mathbf{p}} \mid \mathbf{p} \text{ is a prefix}\}$. Since for any prefix $\mathbf{p}$, the set $\mathcal{S}^\omega_{\mathbf{p}}$ is isomorphic to some closed interval $[a, b]$, we have that Σ_s is isomorphic to the Borel algebra on reals.

Definition 36 (determines ψ on DTMC)**.** *We say a prefix* $\mathbf{p}$ determines *a path formula* ψ *if, for any paths* $\mathbf{x}, \mathbf{x}' \in \mathcal{S}^\omega_{\mathbf{p}}$, $\psi(\mathbf{x}) = \psi(\mathbf{x}')$*; since all paths must have the same value for* ψ *in this case, we denote this quantity as* $\psi(\mathbf{p})$*. Note that any infinite prefix determines* ψ*.*

Definition 37 (Finitely Measurable ψ on DTMC)**.** *We say a path formula* ψ *is finitely measurable on a DTMC structure* D *if for every path* $\pi \in \mathcal{S}^\omega_{\mathbf{p}}$ *with* $\psi(\pi) > 0$*, either:*

1. *there exists a finite prefix* $\mathbf{p}$ *such that* $\psi(\pi) = \psi(\pi')$ *for any paths* $\pi, \pi' \in \mathcal{S}^\omega_{\mathbf{p}}$ *and we denote the quantity by* $\psi(\mathbf{p})$*, or*
2. *the path* π *has probability measure zero.*

Definition 38 (Measure of the Expected Value of ψ on DTMC)**.** *For any finitely measurable formula* ψ*, we define the measure* $\mu_\psi : \mathcal{G}_\mathcal{S} \to \mathbb{R}^*$ *by*

$$\mu_\psi(\mathcal{S}^\omega_{\mathbf{p}}) = \begin{cases} \psi(\pi) \prod_{i=1}^{|\pi|-1} \mathbf{P}(\pi_{i-1}, \pi_i) & \text{if } \pi = (\pi_0, \pi_1, \ldots) \text{ determines } \psi \\ \sum_{s \in \mathcal{S}} \mu_\psi(\mathcal{S}^\omega_{(\mathbf{p},s)}) & \text{otherwise} \end{cases}$$

with $\mu_\psi(\emptyset) = 0$*.*

Also, for the following discussion, we assume a set $\mathcal{AF}$ of atomic state formulas, and a set $\mathcal{AR}$ of atomic restricted state formulas. We are now ready to define CTML.

6.1.2 Syntax of CTML

The syntax of CTML is strongly influenced by PCTL and CTL. As it is mentioned previously, both CTML and PCTL use temporal path operators, namely, "next", "until", and "weak until"; and they both use a superscript $t \in \mathbb{N} \cup \{\infty\}$ that allows "time" related specification for until and weak until path operators. Moreover, neither PCTL nor CTML allow path operators be arbitrarily nested, but rather, they must appear in a pair with a path quantifier for being nested. The main differences between CTML and PCTL are now listed below:

- CTML does not use $\bowtie \in \{<, >, \leq, \geq\}$ operator as a subscript for quantifier P, but rather, it uses $\bowtie$ as a binary operator between two CTML state formulas.
- CTML allows operators $\odot \in \{+, \times\}$ between two state CTML formulas for performing arithmetic operations, which PCTL does not have.
- CTML allows subscript $\odot \in \{+, \times\}$ in until and weak until operators so that these operators can either sum or multiply values along a path until a condition is met.
- CTML allows a single "path quantifier", M, that indicates the expected value (mean) of a path formula, which has no counter part in PCTL.
- CTML has a $1-$ (read as "one minus") unary operator for subtraction from one.
- Among all these syntactic differences, the uttermost one is CTML uses a basic set of atomic function $\mathcal{AF}$, as opposed to a set of atomic propositions $\mathcal{AP}$ used in PCTL.

Definition 39 (CTML Syntax)**.** *Let $f \in \mathcal{AF}, r \in \mathcal{AR}$, and let one, zero denote constant values of* 1 *and* 0*, respectively, the syntax of CTML can be defined recursively as follows:*

$$
\begin{aligned}
\phi &::= f \mid \varphi \mid \phi \odot \phi \mid M\,\psi \\
\varphi &::= one \mid zero \mid r \mid \phi \bowtie \phi \mid \varphi \cdot \varphi \mid 1 - \varphi \mid M\varrho \\
\psi &::= \phi\, U_{\odot}^{\leq t}\, \phi \mid \phi\, V_{\odot}^{\leq t}\, \phi \mid \phi\, U_{+}\, \phi \mid X\,\phi \mid \varphi\, U_{\times}\, \phi \mid \varphi\, V_{\times}\, \phi \\
\varrho &::= \varphi\, U_{\times}^{\leq t}\, \varphi \mid \varphi\, V_{\times}^{\leq t}\, \varphi \mid \varphi\, U_{\times}\, \varphi \mid \varphi\, V_{\times}\, \varphi \mid X\varphi
\end{aligned}
$$

Finally, we note that a "top-level" formula in CTML is a state formula, similar in style to CTL and PCTL, except that we require some "restricted" formulas.

6.1.3 Semantics of CTML

The semantics of CTML are defined in terms of real-valued functions, rather than a satisfaction relation, while still preserving the existing meaning of temporal logics such as PCTL and CTL. We now give the formal semantics of the operators appearing in the language.

Definition 40. *(CTML Semantics) Let f, g be state formulas, then the semantics of CTML can be recursively defined as follows:*

- *If $h = f \bowtie g$, then $h(s) = 1$ if $f(s) \bowtie g(s)$ holds, otherwise $h(s) = 0$.*
- *If $h = f + g$, then $h(s) = f(s) + g(s)$.*
- *If $h = f \cdot g$, then $h(s) = f(s) \cdot g(s)$.*
- *If $h = 1 - f$, then $h(s) = 1 - f(s)$.*
- *If $h = M\psi$, then $h(s) = \mu_\psi(\mathcal{S}^\omega_s)$.*
- *If $\psi = X f$, then $\psi(\pi_0, \pi_1, \ldots) = f(\pi_1)$.*
- *If $\psi = f\, U^{\leq t}_{\odot}\, g$, with $\odot \in \{+, \times\}$ and $t \in \mathbb{N} \cup \{\infty\}$, then*

$$\psi(\pi_0, \pi_1, \ldots) = \begin{cases} \left(\bigodot_{i=0}^{j-1} f(\pi_i)\right) \cdot g(\pi_j) & \text{if } \exists j : 0 \leq j \leq t,\ g(\pi_j) > 0 \text{ and} \\ & \quad \forall 0 \leq i < j, g(\pi_i) = 0 \\ 0 & \text{otherwise} \end{cases}$$

with $f\, U^{\leq \infty}_{\odot}\, g \equiv f\, U_{\odot}\, g$.

- *If* $\psi = f\, V_{\odot}^{\leq t}\, g$*, with* $\odot \in \{+, \times\}$ *and* $t \in \mathbb{N} \cup \{\infty\}$*, then*

$$\psi(\pi_0, \pi_1, \ldots) = \begin{cases} \bigodot_{i=0}^{t} f(\pi_i) & \text{if } \forall 0 \leq i \leq t, g(\pi_i) = 0 \\ \left(\bigodot_{i=0}^{j-1} f(\pi_i)\right) \cdot g(\pi_j) & \text{if } \exists j : 0 \leq j \leq t,\ g(\pi_j) > 0 \text{ and } \\ & \forall 0 \leq i < j, g(\pi_i) = 0 \\ 0 & \text{otherwise} \end{cases}$$

with $f\, V_{\odot}^{\leq \infty}\, g \equiv f\, V_{\odot}\, g$*.*

Now we show that ψ is finitely measurable on the Markov chain as required by $M\psi$. The formula $X f$ has prefix of length 2, and thus is finitely measurable. To see that the path formula $f\, U_{\odot}\, g$ is finitely measurable, consider any path $(\pi_0, \pi_1, \ldots)$ with $\psi(\pi_0, \pi_1, \ldots) > 0$. For this path, there must exist a j satisfying the first condition, namely $j \leq t$, $g(\pi_j) > 0$, and $\forall i < j$, $g(\pi_i) = 0$. Since formula $f\, U_{\odot}^{\leq t}\, g$, has the finite prefix $(\pi_0, \ldots, \pi_j)$, it is finitely measurable. To see that the path formula $f\, V_{\odot}\, g$ is finitely measurable, for the case when t is an integer, then clearly ψ has a finite prefix $(\pi_0, \ldots, \pi_t)$ as of the case for U operator, and thus ψ is finitely measurable. For infinite t, when f is a restricted state formula and $\odot = \times$, we have that $\psi(\mathbf{x})$ is positive for path $\mathbf{x} = (\pi_0, \pi_1, \ldots)$ if and only if the product $f(\pi_0) f(\pi_1) \cdots$ has at most finitely many terms less than one. In a finite DTMC, this can occur either if $g(\pi_j) > 0$ for some j, or if the path contains infinitely many "loops" on states where f evaluates to one. The only such paths with non-zero probability measure are those that eventually reach a recurrent class, where f evaluates to one for every state in the recurrent class. We therefore have that formula $f\, V_{\times}\, g$ is finitely measurable, for restricted state formulas f.

Finally, we can define the measure of the top-level formula ϕ on DTMC structure D based on the initial distribution as follow.

Definition 41 (measure of D_ϕ)**.** *Given a DTMC structure* $D = (\mathcal{S}, \mathbf{P}, \boldsymbol{\pi}_0)$*, the total value of a CTML formula* ϕ *on* D *is defined by*

$$D_\phi = \sum_{s \in \mathcal{S}} \phi(s) \boldsymbol{\pi}_0(s). \tag{6.2}$$

Take the Figure 5.1 as an example, we now show several specification examples. For convenience, in the following, let 0.5 be a (constant) atomic state formula; alternatively, we can use a symbol v as an atomic state formula such that $v(s) = 0.5$ for every state s.

- "With probability at least 0.5, the system will eventually reach $fail$ state before 100 time unit." This query can be specified by both PCTL and CTML as shown below.

$$\text{PCTL:} \quad P_{\geq 0.5}\, true\, U^{\leq 100}\, fail$$

$$\text{CTML:} \quad (M\, one\, U_{\times}^{\leq 100}\, fail) \geq 0.5$$

- "With probability at least 0.5, the system will eventually reach a $trying$ state, and since then it will keep trying until $fail$" This query cannot be expressed by PCTL, but can be expressed by both PLTL and CTML as shown below:

$$\text{PLTL:} \quad P_{\geq 0.5} F(trying \wedge trying\, U fail)$$

$$\text{CTML:} \quad M\, one\, U_{\times}(trying \cdot M\, trying\, U_{\times}\, fail)) \geq 0.5$$

- "What is the expected amount of time that the system will be keep trying before it reaches the $succ$ state?" This query can neither be expressed by PCTL, nor by PLTL, but can be expressed by CTML as shown below:

$$\text{CTML:} \quad M\, trying\, U_{+}\, succ$$

  Since this query is under conditional distribution, according to the conditional expectation definition 2.5, the result must be divided by the quantity of $M\, trying\, U_{\times}\, succ$.

- "Given that a process just reached the trying state, then what is the expected time until failure?" First, we determine the expected time until fail starting from each possible state. Then, we filter out all but the states where the process has been trying. We then sum over all trying states, the probability to reach that one "first" multiplied by the expected time to failure starting from that

state. This gives us the expression

$$\text{CTML:} \quad M\,one\,U_{\times}((M\,one\,U_{+}\,fail)\cdot trying).$$

The final quantity can be obtained using conditional probability law, which says that $\Pr\{B|A\} = \frac{\Pr\{B\cap A\}}{\Pr\{A\}}$. In this case, we just divide the result by $M\,one U_{\times}\,trying$.

To our best knowledge, this type of query cannot be expressed by any existing (formal) stochastic logics such as CSL [6], CSRL [7], DTRMC [2], CSL$^{\text{TA}}$ [29], etc., either because they do not operate on real values (such as CSL, CSL$^{\text{TA}}$), or because they cannot nest real values (such as CSRL, DTRMC).

6.2 Algorithms

Given the CTML language definition, the computation of algebraic formulas such as $\phi + \phi$, $\phi \cdot \phi$, $1 - \phi$, $\phi \bowtie \phi$, etc. can be carried out directly by their semantics. In this section, we present algorithms for computing CTML formulas, namely MX, $MU_{\times}^{\leq t}$, $MU_{+}^{\leq t}$, $MV_{\times}^{\leq t}$, and $MV_{+}^{\leq t}$, where $t \in \mathbb{N} \cup \{\infty\}$. When t is unbounded, the algorithms for bounded t may not be feasible due to possibly infinite computation. For that reason, we give different algorithms for bounded time t and unbounded time t. The state formula for $h = MX\,f$ is given by $h(s) = \sum_{s_1 \in \mathcal{S}} f(s_1)\mathbf{P}[s, s_1]$, in this case the vector $\mathbf{h}$ can be determined simply as $\mathbf{h} = \mathbf{Pf}$, which is similar to PCTL in style, except the vector $\mathbf{f}$ can contain real-valued quantities, rather than merely one or zero.

6.2.1 Algorithms For The Case of MU

To compute $\mathbf{h} = M\psi, \psi = fU_{\odot}^{\leq t}\,g$, we first modify the model as follows:

1. states s_g for which $g(s_g) > 0$ are made absorbing, and this set is grouped into $\mathcal{S}_g$ (called success states);

2. states s_n from which it is impossible to reach a state in $\mathcal{S}_g$ are also made absorbing, and this set is grouped into $\mathcal{S}_n$ (called failure states). Note that the formula evaluates to zero for this set of states.

3. the remaining states are all transient states, and grouped into $\mathcal{S}_z = \mathcal{S} \setminus (\mathcal{S}_g \cup \mathcal{S}_n)$.

The model modification uses a similar approach to PCTL, but here the set $\mathcal{S}_g$ has formula g that requires to be evaluated to a positive real value, rather than value of one (i.e. true in PCTL).

Note that it can be shown that the state formula $M\psi$ evaluated on $(\mathcal{S}, \mathbf{P}, \boldsymbol{\pi}_0)$ is equivalent to $M\psi$ evaluated on $(\mathcal{S}, \mathbf{P}', \boldsymbol{\pi}_0)$: any path π with $\psi(\pi) > 0$ must contain a state π_j with $g(\pi_j) > 0$, and since the prefix $(\pi_0, \ldots, \pi_j)$ determines ψ, all that remains is to demonstrate that measure $\mu_\psi(\mathcal{S}^\omega_{\mathbf{p}})$ gives the same value on $\mathbf{P}'$ as it does on $\mathbf{P}$, which follows from the fact that rows of $\mathbf{P}'$ and $\mathbf{P}$ are equal for states in $\mathcal{S}_z$. The benefit of this modification is that, under $\mathbf{P}'$, for any path $\mathbf{p} = (\pi_0, \pi_1, \ldots)$ such that $\psi(\mathbf{p}) > 0$, it must be the case that at time t, $\pi_t \in \mathcal{S}_g$.

For the following discussions, we assume $\mathbf{P}$ has already been modified for operator U. $\mathbf{f}$ denotes a corresponding vector of state formula f, $\mathbf{g}$ denotes a corresponding vector of state formula g, and $\mathbf{F} = diag(\mathbf{f})$.

6.2.1.1 Case of MU_+

In this case, first, we modify f such that $f(s_g) = 0$ for all $s_g \in \mathcal{S}_g$, and keep $g(s)$ as is for all $s \in \mathcal{S}$. Following the semantics of the until with addition formula, for the state formula of $h_t = Mf\,U_+^{\leq t}\,g$, we have:

$$h_t(s_0) = \sum_{(s_1,\ldots,s_t)\in\mathcal{S}^t} (f(s_0) + \cdots + f(s_{t-1}))\,\mathbf{P}[s_0, s_1]\cdots\mathbf{P}[s_{t-1}, s_t]g(s_t) \tag{6.3}$$

$$= \sum_{(s_1,\ldots,s_t)\in\mathcal{S}^t} f(s_0)\mathbf{P}[s_0, s_1]\cdots\mathbf{P}[s_{t-1}, s_t]g(s_t) + \sum_{s_1\in\mathcal{S}} \mathbf{P}[s_0, s_1]h_{t-1}(s_1) \tag{6.4}$$

where the recurrence terminates with $h_0(s) = 0$ for all s.

The recurrence (6.4) can be written as the matrix equation

$$\mathbf{h}_t = \mathbf{F}\mathbf{P}^t\mathbf{g} + \mathbf{P}\mathbf{h}_{t-1} \tag{6.5}$$

with terminating condition $\mathbf{h}_0 = \mathbf{0}$.

In practice, we want to avoid computing $\mathbf{P}^t$ directly. So, for finite t, (6.5) can be computed using auxiliary vector $\boldsymbol{\pi}_t = \mathbf{P}^t\mathbf{g} = \mathbf{P}\boldsymbol{\pi}_{t-1}$ with $\boldsymbol{\pi}_0 = \mathbf{g}$.

For unbounded until, the state formula for $h = Mf\,U_+\,g$ is given by $h = \lim_{t\to\infty} Mf\,U_+^{\leq t}\,g$. Defining $\mathbf{h} = \lim_{t\to\infty}\mathbf{h}_t$, from (6.5) we obtain the linear system $(\mathbf{I}-\mathbf{P})\mathbf{h} = \mathbf{F}(\lim_{t\to\infty}\mathbf{P}^t)\mathbf{g}$. Since $\mathbf{P}$ is an absorbing DTMC, $\lim_{t\to\infty}\mathbf{P}^t$ is given by (3.7). Now, let $\mathbf{y} = \lim_{t\to\infty}\mathbf{P}^t\mathbf{g}$. Note that $\mathbf{y}[i]$ is equal to the probability that the DTMC eventually reaches a state in $\mathcal{S}_g$, starting from state i. We can split the system based on $\mathcal{S} = \mathcal{S}_z \cup \mathcal{S}_n \cup \mathcal{S}_g$ to obtain

$$\begin{bmatrix} \mathbf{y}(\mathcal{S}_z) \\ \mathbf{y}(\mathcal{S}_n) \\ \mathbf{y}(\mathcal{S}_g) \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{NP}[\mathcal{S}_z,\mathcal{S}_n] & \mathbf{NP}[\mathcal{S}_z,\mathcal{S}_g] \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{g}(\mathcal{S}_g) \end{bmatrix}$$

and the solution of this system gives $\mathbf{y}(\mathcal{S}_z) = \mathbf{NP}[\mathcal{S}_z,\mathcal{S}_g]\mathbf{g}(\mathcal{S}_g)$, $\mathbf{y}(\mathcal{S}_n) = \mathbf{0}$, and $\mathbf{y}(\mathcal{S}_g) = \mathbf{g}(\mathcal{S}_g)$. Vector $\mathbf{y}(\mathcal{S}_z)$ can be obtained by solving the linear system

$$(\mathbf{I} - \mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])\mathbf{y}(\mathcal{S}_z) = \mathbf{P}[\mathcal{S}_z,\mathcal{S}_g]\mathbf{g}(\mathcal{S}_g). \tag{6.6}$$

Since $\mathbf{h}[s]$ is zero for $s \in \mathcal{S}_n \cup \mathcal{S}_g$, the linear system for $\mathbf{h}$ becomes

$$\begin{bmatrix} \mathbf{I} - \mathbf{P}[\mathcal{S}_z,\mathcal{S}_z] & \mathbf{P}[\mathcal{S}_z,\mathcal{S}_n] & \mathbf{P}[\mathcal{S}_z,\mathcal{S}_g] \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{h}(\mathcal{S}_z) \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{F}(\mathcal{S}_z,\mathcal{S}_z) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}(\mathcal{S}_n,\mathcal{S}_n) & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{y}(\mathcal{S}_z) \\ \mathbf{0} \\ \mathbf{g}(\mathcal{S}_g) \end{bmatrix}, \text{ which produces a single equation}$$

$$(I - \mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])\mathbf{h}(\mathcal{S}_z) = \mathbf{F}(\mathcal{S}_z,\mathcal{S}_z)\mathbf{y}(\mathcal{S}_z). \tag{6.7}$$

Therefore, we can obtain $\mathbf{h}$ by first solving (6.6) for $\mathbf{y}$, and then solving (6.7).

6.2.1.2 Case of $MU_{\times}$

Similar to until with addition, assume that the model has already been modified, with $\mathcal{S} = \mathcal{S}_g \cup \mathcal{S}_n \cup \mathcal{S}_z$, and $\mathbf{P}$ has already been modified for U accordingly as well. Now we modify f such that $f(s_g) = 1$ for $s_g \in \mathcal{S}_g$, and keep g as is. Following the semantics for until with multiplication, the state formula for $h_t = Mf\, U_{\times}^{\leq t}\, g$ is given by

$$h_t(s_0) = \sum_{(s_1,\ldots,s_t)\in\mathcal{S}^t} f(s_0)\cdots f(s_{t-1})\mathbf{P}[s_0,s_1]\cdots\mathbf{P}[s_{t-1},s_t]g(s_t) \tag{6.8}$$

$$= \sum_{(s_1,\ldots,s_t)\in\mathcal{S}^t} f(s_0)\mathbf{P}[s_0,s_1]h_{t-1}(s_1) \tag{6.9}$$

where the recurrence terminates with $h_0(s) = g(s)$.

Now, let $Mult(t,s)$ denote the measure of expected value for the set of paths quantified by path formula $f\, U_{\times}^{\leq t}\, g$, with respect to starting state s, then the following pseudocode gives more intuitive algorithm for computing recurrence (6.9) discussed above.

Algorithm 6.1 Compute $Mult(t,s)$

if $t = 0$ **then**
  return $g(s)$
end if

if $g(s) = 0$ || $f(s) = 0$ **then**
  return 0
end if

return $\Sigma_{s'\in\mathcal{S}} f(s)\cdot\mathbf{P}[s,s']\cdot Mult(t-1,s')$

Alternatively, recurrence (6.9) can be written as the matrix equation

$$\mathbf{h}_t = \mathbf{FP}\mathbf{h}_{t-1} \tag{6.10}$$

with terminating condition $\mathbf{h}_0 = \mathbf{g}$. For a finite t, $\mathbf{h}_t$ can be computed iteratively. Note that unrolling the recurrence gives us $\mathbf{h}_t = (\mathbf{FP})^t\mathbf{g}$.

For unbounded until, we obtain $\mathbf{h} = \lim_{t\to\infty}(\mathbf{FP})^t\mathbf{g}$. If no value of f is greater than one, then it can be shown that $\lim_{n\to\infty}(\mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])^n = \mathbf{0}$, where $\mathbf{F}_z$ is shorthand for $\mathbf{F}(\mathcal{S}_z,\mathcal{S}_z)$, since $\mathbf{P}$ is the transition probability matrix of an absorbing DTMC with transient states $\mathcal{S}_z$. It then follows that

$$(\mathbf{I} - \mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])\sum_{i=0}^{\infty}(\mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])^i = \mathbf{I}$$

and therefore $(\mathbf{I}-\mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])^{-1}$ exists. Looking at the solution vector $\mathbf{h}$, if we again split the system based on $\mathcal{S} = \mathcal{S}_z \cup \mathcal{S}_n \cup \mathcal{S}_g$, we can obtain

$$\begin{aligned}\mathbf{h}[\mathcal{S}_z] &= \lim_{n\to\infty}\left(\sum_{i=0}^{n}(\mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])^i\right)\mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_g]\mathbf{g}(\mathcal{S}_g)\\ &= (\mathbf{I} - \mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])^{-1}\mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_g]\mathbf{g}(\mathcal{S}_g),\end{aligned}$$

while $\mathbf{h}[\mathcal{S}_n] = \mathbf{0}$ and $\mathbf{h}[\mathcal{S}_g] = \mathbf{g}(\mathcal{S}_g)$. This implies that the linear system

$$(\mathbf{I} - \mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_z])\mathbf{h}[\mathcal{S}_z] = \mathbf{F}_z\mathbf{P}[\mathcal{S}_z,\mathcal{S}_g]\mathbf{g}(\mathcal{S}_g) \tag{6.11}$$

can be solved to obtain $\mathbf{h}[\mathcal{S}_z]$.

6.2.2 Algorithm For the Case of MV

To compute $h = Mf\,V_{\odot}^{\leq t}\,g$, we use a slightly modified transition probability matrix $\mathbf{P}'$ similar to the case for U in the sense that the states s_g for which $g(s_g) > 0$ are made absorbing, and we denote this set of states as $\mathcal{S}_g$. By the semantics of the path formula $f\,V_{\odot}^{\leq t}\,g$, if there is no s_g with $g(s_g) > 0$, then f is summed up or multiplied up to the t^{th} state, for a given path. So in this case, there is no need to identify and partition the other two sets of states namely $\mathcal{S}_z$ and $\mathcal{S}_n$ as the case of until.

6.2.2.1 Case of MV_+

We assume that $\mathbf{P}$ has already been modified for V as discussed above, and that $f(s_g) = 0$ for $s_g \in \mathcal{S}_g$. Additionally, after constructing $\mathcal{S}_g$, we modify g such that all zero values are replaced with

one. With these modifications, the state formula for $h_t = Mf\, V_+^{\leq t}\, g$ is given by

$$h_t(s_0) = \sum_{(s_1,\ldots,s_t)\in\mathcal{S}^t} (f(s_0) + \cdots + f(s_t))\, \mathbf{P}[s_0, s_1] \cdots \mathbf{P}[s_{t-1}, s_t] g(s_t) \tag{6.12}$$

where the recurrence terminates with $h_0(s) = f(s)$ for all s. Since (6.12) is identical to (6.3), we obtain the same matrix equation as strong until with addition, $\mathbf{h}_t = \mathbf{F}\mathbf{P}^t\mathbf{g} + \mathbf{P}\mathbf{h}_{t-1}$, except we use a different terminating vector $\mathbf{h}_0$.

CTML does not currently support V_+ in the unbounded case, because such a formula produces values of infinity whenever there is a recurrent class $\mathcal{C}$ with $\mathcal{S}_g \cap \mathcal{C} = \emptyset$, and $f(s) > 0$ for some $s \in \mathcal{C}$. While these cases are easy to determine, issues with infinity in the *language* must be resolved (e.g., determining values for $\infty \geq \infty$ or $\infty \cdot 0$) before CTML can support unbounded V_+.

6.2.2.2 Case of $MV_\times$

Again, assume that $\mathbf{P}$ has already been modified for V as discussed above, except we can add states with $f(s) = 0$ and $g(s) = 0$ to the set $\mathcal{S}_n$. Additionally, we modify f such that $f(s_g) = 1$ for $s_g \in \mathcal{S}_g$, and after these modifications are complete, we modify g so that all zero values are replaced with the value one. Then, the state formula for $h_t = Mf\, V_\times^{\leq t}\, g$ becomes

$$h_t(s_0) = \sum_{(s_1,\ldots,s_t)\in\mathcal{S}^t} f(s_0) \cdots f(s_t) \mathbf{P}[s_0, s_1] \cdots \mathbf{P}[s_{t-1}, s_t] g(s_t) \tag{6.13}$$

where the recurrence terminates with $h_0(s) = f(s) \cdot g(s)$ for all s. Since (6.13) is identical to (6.8), we obtain the same matrix equation as strong until with multiplication, $\mathbf{h}_t = \mathbf{F}\mathbf{P}\mathbf{h}_{t-1}$, but with terminating condition $\mathbf{h}_0 = \mathbf{F}\mathbf{g}$.

Alternatively, let WeakMult(t, s) denote the measure of expected value for the set of paths quantified by path formula $f\, V_\times^{\leq t}\, g$, with respect to starting state s, then the following pseudocode gives more intuitive algorithm for computing recurrence (6.13) discussed above.

For unbounded weak until, consistent with the discussion in Definition 40, the state formula $h = Mf\, V_\times\, g$ is equivalent to the state formula $h' = Mf\, U_\times\, g'$ where $g'(s)$ has value $g(s)$ if $g(s) > 0$, has value one if s is in a recurrent class where f evaluates to one for every state in the recurrent class, and

Algorithm 6.2 Compute WeakMult(t, s)

if $t = 0$ **then**
  return $f(s)g(s)$
end if

if $g(s) = 0$ || $f(s) = 0$ **then**
  return 0
end if

return $\Sigma_{s' \in \mathcal{S}} f(s) \cdot \mathbf{P}[s, s'] \cdot \text{WeakMult}(t-1, s')$

has value zero otherwise. Therefore, an algorithm for $MV_\times$ is to determine the recurrent classes (after making states with $g(s) > 0$ absorbing), set the function g' appropriately, and invoke the algorithm for $MU_\times$.

6.2.3 Computational Complexity

For CTML, the overall computational complexity is $O(|\phi| \cdot Poly(|\mathcal{S}|))$, where $|\phi|$ counts the number of operators in formula ϕ, and $Poly(|\mathcal{S}|)$ denotes polynomial time in the size of $|\mathcal{S}|$. This is similar to the complexity for PCTL [33], but better than the $O(2^{|\phi|} \cdot Poly(|\mathcal{S}|))$ complexity to compute the probability for a PLTL formula [22]. Specifically, assuming we limit our computations to floating-point or rational values, the trivial operations $f \bowtie g$ and $f \odot g$ can be computed in $O(|\mathcal{S}|)$ time. The operation MXf requires matrix-vector multiplication, which can be done in $O(|D|)$ time, where $|D|$ refers to the size of the DTMC (the number of non-zero entries in matrix $\mathbf{P}$ plus the number of states). The bounded operators $MfU_{\odot}^{\leq t}g$ and $MfV_{\odot}^{\leq t}g$ can be computed in $O(t \cdot |D|)$ time, and the unbounded operators $MfU_{\odot}^{\leq t}g$ and $MfV_\times g$ can be computed using solution of linear systems, which requires $O(Poly(|\mathcal{S}|))$. Therefore, assuming t is at most polynomial in $|\mathcal{S}|$, the value of a CTML formula ϕ can be determined in $O(|\phi| \cdot Poly(|\mathcal{S}|))$ time.

CHAPTER 7. COMPARING CTML'S EXPRESSIVE POWER WITH PCTL AND PLTL

In the previous chapter, we introduced CTML syntax, semantics, algorithms, and proofs. We mentioned that the expressive power of CTML, compared with the existing stochastic logics, are in two aspects: 1) it can express real values from 0 to infinity; 2) it can nest real values. While it is obvious that neither PLTL nor PCTL can express general real values that CTML can express, it is not clear that how much a strict subset of CTML, alone, can express for the corresponding PLTL formulas and/or PCTL formulas.

In this chapter, we investigate the expressibility results of a strict subset of CTML, namely, the restricted sublanguage of CTML (where the real values is limited to the range of $[0, 1]$), by comparing it to PLTL and PCTL.

7.1 CTML vs. PLTL

In terms of probability measures, rather than general real valued quantities, one obvious difference between CTML and PLTL is that PLTL cannot express probability measures of path formulas nested with time bounded operators such as $Mone\,\mathrm{U}_{\times}(Mleft\,\mathrm{U}_{\times}^{\leq t}eat)$. The second advantage of CTML is that its overall computational complexity of CTML is polynomial in the size of both the specification formula and the input model, whereas PLTL requires exponential time in the size of specification formula and polynomial in the size of the model structure [22]. For example, if an LTL formula consists of n until operators in right recursion such as $a_1\mathrm{U}(a_2\mathrm{U}(\cdots(a_{n-1}\mathrm{U}a_n)))$, then it requires computational complexity of $O(2^n|M|)$, where $|M|$ denotes the size of the model. In this section, we show a strict sublanguage of CTML, in which a state formula is restricted to the real value of $[0, 1]$, that can express a sublanguage of PLTL characterized by right recursion.

Note that for the following discussion, when the context is clear, we use the same atomic symbols such as $a, b, \cdots$, for both CTML and PLTL. Let L be the labeling function for LTL expressions. Let $\Pr_s^D(\Psi)$ denote the probability that LTL path formula Ψ holds, and $\phi^D(s)$ denote the value of CTML state formula ϕ, starting from state s in DTMC D. When the context is clear, we use $\Pr_s(\Psi)$ rather than $\Pr_s^D(\Psi)$, and $\phi(s)$ rather than $\phi^D(s)$.

Lemma 42. *Let Ψ be an LTL path formula, and ϕ be a CTML state formula, such that for any DTMC structure D and any starting state s in D, $\Pr_s(\Psi) = \phi(s)$. Then,*

$$\Pr_{s'}(\mathrm{X}\Psi) = (M\mathrm{X}\phi)(s'), \textit{ for all } s' \in \mathcal{S}.$$

Proof. Suppose $\Pr_s(\Psi) = \phi(s)$ for all $s \in \mathcal{S}$, then by LTL's semantics, we have $\pi \models \mathrm{X}\Psi$ iff $\pi^1 \models \Psi$. Let $\Pr(\pi^1)$ denote the probability for the path $\pi^1 \models \Psi$. Let π_0^1 denote the starting state for the suffix π^1. Then, we have

$$\begin{aligned}
\Pr_{s'}(\mathrm{X}\Psi) &= \sum_{\substack{\pi \models X\Psi \\ \pi_0 = s'}} \Pr(\pi) \quad \text{by PLTL's semantics} \\
&= \sum_{\pi_1 \in \mathcal{S}} \sum_{\substack{\pi_0^1 = \pi_1 \\ \pi^1 \models \Psi}} \mathbf{P}[s', \pi_1] \cdot \Pr(\pi^1) \\
&= \sum_{\pi_1 \in \mathcal{S}} \mathbf{P}[s', \pi_1] \cdot \sum_{\substack{\pi_0^1 = \pi_1 \\ \pi^1 \models \Psi}} \Pr(\pi^1) \\
&= \sum_{\pi_1 \in \mathcal{S}} \mathbf{P}[s', \pi_1] \cdot \phi(\pi_1) \quad \text{derived CTML semantics} \\
&= (M\mathrm{X}\phi)(s')
\end{aligned}$$

Therefore, we have $\Pr_{s'}(\mathrm{X}\Psi) = (M\mathrm{X}\phi)(s')$ holds for all $s' \in \mathcal{S}$.

□

Lemma 43. *Let Ψ be an LTL path formula, and ϕ be a CTML state formula, such that for any DTMC structure D and any starting state s in D, $\Pr_s(\Psi) = \phi(s)$. Then for any atomic symbol a,*

$$\Pr_{s'}(a \,\mathrm{U}\, \Psi) = (Ma \,\mathrm{U}_{\times}\, \phi)(s'), \textit{ for all } s' \in \mathcal{S}.$$

Proof. Suppose $\Pr_s(\Psi) = \phi(s)$ for all $s \in \mathcal{S}$. Then by LTL's semantics, we have $\pi \models a \,\mathrm{U}\, \Psi$ iff $\pi^s \models \Psi$, and before that, we have $\pi^0 \models a, \ldots, \pi^{s-1} \models a$; that is, it requires the suffix π^s that satisfy Ψ, and π^s must be preceded by a sequence of 0 or more number of symbol a. Let $\Pr(\pi^s)$ denote the probability of π^s. Let π^i_0 denote the starting state of the suffix π^i. Let ξ^s denote the set of path segments $(\pi_0, \ldots, \pi_s)$ such that $\pi^i \models a, 0 \leq i < s, \pi^i_0 = \pi_i, \pi^s \models \Psi, \pi_s = \pi^s_0$. Then by PLTL's semantics, we have

$$\begin{aligned}
\Pr_{s'}(a \,\mathrm{U} \Psi) &= \sum_{(s',\ldots,\pi_s)\in\xi^s} \sum_{\substack{\pi^s_0=\pi_s \\ \pi^s\models\Psi}} \mathbf{P}[s',\pi_1]\cdot\cdots\cdot\mathbf{P}[\pi_{s-1},\pi_s]\cdot\Pr(\pi^s) \quad \text{by PLTL semantics} \\
&= \sum_{(s',\ldots,\pi_s)\in\xi^s} \mathbf{P}[s',\pi_1]\cdot\cdots\cdot\mathbf{P}[\pi_{s-1},\pi_s] \sum_{\substack{\pi_s=\pi^s_0 \\ \pi^s\models\Psi}} \Pr(\pi^s) \\
&= \sum_{(s',\ldots,\pi_s)\in\xi^s} \mathbf{P}[s',\pi_1]\cdot\cdots\cdot\mathbf{P}[\pi_{s-1},\pi_s]\cdot\phi(\pi_s) \quad \text{derived CTML semantics} \\
&= (Ma \,\mathrm{U}_{\times}\, \phi)(s')
\end{aligned}$$

Therefore, we have $\Pr_{s'}(a \,\mathrm{U}\, \Psi) = (Ma \,\mathrm{U}_{\times}\, \phi)(s')$ holds for all $s' \in \mathcal{S}$.

□

Lemma 44. *Let Ψ be a LTL path formula , and ϕ be a CTML state formula, such that for any DTMC structure D and any starting state s in D, $\Pr_s(\Psi) = \phi(s)$. Also, let a be a LTL atomic proposition and a' be a CTML atomic formula such that for a given path π in D $\pi^0 \models a$ iff $a'(\pi_0) = 1$ and $\pi^0 \not\models a$ iff $a'(\pi_0) = 0$. Then,*

$$\Pr_s(a \wedge \Psi) = (a' \cdot \phi)(s), \textit{ for all } s \in \mathcal{S}.$$

Proof. Suppose $\Pr_s(\Psi) = \phi(s)$ for all $s \in \mathcal{S}$, then by LTL's semantics, we have $\pi \models a \wedge \Psi$ iff $\pi^0 \models \Psi$, and $\pi^0 \models a$. By PLTL's semantics, since a is an atomic symbol, $\Pr_s(a \wedge \Psi) = 1 \cdot \Pr_s(\Psi)$ if a holds on s, and 0 otherwise. By CTML's semantics, $(a' \cdot \phi)(s) = a'(s) \cdot \phi(s) = \phi(s)$ if $a'(s) = 1$ and 0 otherwise. Therefore, we have $\Pr_s(a \wedge \Psi) = (a' \cdot \phi)(s),$ for all $s \in \mathcal{S}$ for all $s \in \mathcal{S}$.

□

Lemma 45. *Let Ψ be an LTL path formula, and ϕ be a CTML state formula, such that for any DTMC structure D and any starting state s in D, $\Pr_s(\Psi) = \phi(s)$. Then,*

$$\Pr_s(\neg\Psi) = 1 - \phi(s), \textit{ for all } s \in \mathcal{S}.$$

Proof. Suppose $\Pr_s(\Psi) = \phi(s)$ for all $s \in \mathcal{S}$, then by LTL's semantics, we have $\pi \models \neg\Psi$ iff $\pi \not\models \Psi$ starting from state s. By the probability measure definition, we have $\Pr_s(\neg\Psi) = 1 - \Pr_s(\Psi) = 1 - \phi(s)$.

□

Definition 46 (Right-recursive LTL Formula)**.** *Let a be any atomic symbol, then the* right recursive LTL formula Ψ' *can be defined by the following grammar:*

$$\Psi' ::= a \mid \mathrm{X}\Psi' \mid a \wedge \Psi' \mid \neg\Psi' \mid a\mathrm{U}\Psi'$$

We now present a translation algorithm, called $translate(\Psi')$, that takes a right recursive LTL formula Ψ' as input, and returns a translated CTML formula ϕ, for $\Pr(\Psi')$.

For example, given $\Psi' = a\mathrm{U}(\, b\,\mathrm{U}\,(\, c \wedge d\,))$, translate$(\Psi')$ can go recursively as follows:

1. case $a\mathrm{U}\Psi' : \phi = Ma\mathrm{U}_\times$ (translate$(b\mathrm{U}(c \wedge d))$);
2. case $b\mathrm{U}\Psi' : \phi = Ma\mathrm{U}_\times(Mb\mathrm{U}_\times$ (translate $(c \wedge d)))$
3. case $c \wedge \Psi' : \phi = Ma\mathrm{U}_\times(Mb\mathrm{U}_\times(c\cdot$ (translate $(d))))$
4. case $d : \phi = Ma\mathrm{U}_\times(Mb\mathrm{U}_\times(c \cdot d))$.

Algorithm 7.1 translate (Ψ')

Input: Ψ'
Output: ϕ
  switch Ψ' **do**
    case a
      return a
    case $\mathrm{X}\Psi'$
      return $M\mathrm{X}$ translate (Ψ')
    case $a\mathrm{U}\Psi'$
      return $Ma\mathrm{U}_{\times}$ translate (Ψ')
    case $a \wedge \Psi'$
      return $a\cdot$ translate (Ψ')
    case $\neg\Psi'$
      return $1-$ translate (Ψ')

Theorem 47 $(\Pr(\Psi') = \phi)$**.** *Given any right-recursive LTL formula Ψ', it is expressible in CTML with $\Pr_s(\Psi') = \phi(s)$, for a given DTMC structure D and starting state s, where $\phi =$ translate(Ψ') based on the translation Algorithm 7.1.*

Proof. In case a is an atomic symbol, we have $Pr_s(a) = a(s)$ for all $s \in \mathcal{S}$ trivially. For all other cases, it directly follows from Lemma 42, Lemma 43, Lemma 44, and Lemma 45. Hence, we have one to one mapping from the right recursive PLTL formula $\Pr_s(\Psi')$ to CTML formula $\phi'(s)$. □

7.2 CTML vs. PCTL

Theorem 48. *For any PCTL state formula ϕ, there exists a CTML state formula ϕ' such that, for any DTMC and any state s, $s \models \phi$ iff $\phi'(s) = 1$ and $s \not\models \phi$ iff $\phi'(s) = 0$.*

Proof. Proof by induction on the structure of PCTL grammar.

- **Base case:** Given a PCTL atomic proposition $p \in \mathcal{AP}$, there is a restricted CTML atomic function $p' \in \mathcal{AF}$ such that $p'(s) = 1$ iff $s \models p$, and $p'(s) = 0$ iff $s \not\models p$.
- **Inductive case:** If ϕ, ϕ_1, ϕ_2 are PCTL state formulas with CTML equivalents ϕ', ϕ_1', ϕ_2' such that $s \models \phi, s \models \phi_1, s \models \phi_2$ iff $\phi'(s) = 1, \phi_1'(s) = 1$, and $\phi_2'(s) = 1$, respectively, and $s \not\models \phi, s \not\models \phi_1, s \not\models \phi_2$ iff $\phi'(s) = 0, \phi_1'(s) = 0$, and $\phi_2'(s) = 0$, respectively. Then,

– $s \models \neg\phi$ iff $s \not\models \phi$ iff $\phi'(s) = 0$ iff $\phi' < one(s) = 1$; and $s \not\models \neg\phi$ iff $s \models \phi$ iff $\phi'(s) = 1$ iff $\phi' < one(s) = 0$.

Given $s \models \neg\phi$ iff $s \not\models \phi$, we have $\phi'(s) = 0$, so $\phi'(s) < one(s)$ holds in CTML, hence $\phi' < one(s) = 1$. On the other hand, if $\phi' < one(s) = 1$ in CTML, then this implies $\phi'(s) = 0$, which implies $s \not\models \phi$, therefore $s \models \neg\phi$ in PCTL.

If $s \not\models \neg\phi$, then it implies $s \models \neg\neg\phi = s \models \phi$. So we have $\phi'(s) < one(s)$ does not hold, which implies $\phi'(s) < one(s) = 0$ and $\phi'(s) = 1$ in CTML. On the other hand, if $\phi' < one(s) = 0$ in CTML, then this directly translates to $s \not\models \neg\phi$ by the above case analysis.

– $s \models \phi_1 \wedge \phi_2$ iff $\phi'_1 \cdot \phi'_2(s) = 1$; and $s \not\models \phi_1 \wedge \phi_2$ iff $\phi'_1 \cdot \phi'_2(s) = 0$;

If $s \models \phi_1$ and $s \models \phi_2$, then $\phi'_1(s) = 1$, $\phi'_2(s) = 1$, then we have $\phi'_1(s) \cdot \phi'_2(s) = 1$, so $\phi'_1 \cdot \phi'_2(s) = 1$. If $\phi'_1 \cdot \phi'_2(s) = 1$, then we must have $\phi'_1(s) = 1$ and $\phi'_2(s) = 1$, which implies $s \models \phi_1$ and $s \models \phi_2$; therefore, $s \models \phi_1 \wedge \phi_2$.

If $s \not\models \phi_1 \wedge \phi_2$, then this implies $s \not\models \phi_1$ or $s \not\models \phi_2$, or both $s \not\models \phi_1$ and $s \not\models \phi_2$; this implies we have $\phi'_1(s) = 0$, or $\phi'_2(s) = 0$, or both $\phi'_1(s) = 0$ and $\phi'_2(s) = 0$, so $\phi'_1 \cdot \phi'_2(s) = 0$. If $\phi'_1 \cdot \phi'_2(s) = 0$, then we must have either $\phi'_1(s) = 0$, or $\phi'_2(s) = 0$, or both $\phi'_1(s) = 0$ and $\phi'_2(s) = 0$; this implies $s \not\models \phi_1$ or $s \not\models \phi_2$; therefore, $s \not\models \phi_1 \wedge \phi_2$.

– $s \models \phi_1 \vee \phi_2$ iff $\phi'_1 + \phi'_2 > zero(s) = 1$; and $s \not\models \phi_1 \vee \phi_2$ iff $\phi'_1 + \phi'_2(s) = 0$.

If $s \models \phi_1 \vee \phi_2$, then we have $s \models \phi_1$, or $s \models \phi_2$, or $s \models \phi_1$, and $s \models \phi_2$; this implies $\phi'_1(s) = 1$, $\phi'_2(s) = 0$, or $\phi'_1(s) = 0$, $\phi'_2(s) = 1$, or $\phi'_1(s) = 1$, $\phi'_2(s) = 1$; whichever the case, we have $(\phi'_1 + \phi'_2) > zero(s) = 1$. On the other hand, if $(\phi'_1 + \phi'_2) > zero(s) = 1$, then we have either $\phi'_1(s) = 1$, or $\phi'_2(s) = 1$, or both; then, we have $s \models \phi_1 \vee \phi_2$.

If $s \not\models \phi_1 \vee \phi_2$, then we have $s \not\models \phi_1$ and $s \not\models \phi_2$; this implies $\phi'_1(s) = 0$ and $\phi'_2(s) = 0$, hence $\phi'_1 + \phi'_2(s) = 0$. If $(\phi'_1 + \phi'_2) > zero(s) = 0$, then it must be $\phi'_1(s) = 0$ and $\phi'_2(s) = 0$; this implies $s \not\models \phi_1$ and $s \not\models \phi_2$, hence $s \not\models \phi_1 \vee \phi_2$.

– $\pi \models X\phi$ iff $X\phi'(\pi) = 1$; and $\pi \not\models X\phi$ iff $X\phi'(\pi) = 0$.

If $\pi \models X\phi$, then by its semantics in PCTL, we have $\pi_1 \models \phi$, which is equivalent to $X\phi'(\pi) = \phi'(\pi_1) = 1$ in CTML. If $X\phi'(\pi) = \phi'(\pi_1) = 1$, then it is equivalent to $\pi_1 \models \phi$

in PCTL, then we have $\pi \models X\phi$. If $\pi \not\models X\phi$, then by its semantics in PCTL, we have $\pi_1 \not\models \phi$, which is equivalent to $X\phi'(\pi) = \phi'(\pi_1) = 0$ in CTML. If $X\phi'(\pi) = \phi'(\pi_1) = 0$, then it is equivalent to $\pi_1 \not\models \phi$ in PCTL, hence, $\pi \not\models X\phi$.

– $\pi \models \phi_1 \, U^{\leq t} \, \phi_2$ iff $\phi_1' \, U_{\times}^{\leq t} \, \phi_2'(\pi) = 1$; and $\pi \not\models \phi_1 \, U^{\leq t} \, \phi_2$ iff $\phi_1' \, U_{\times}^{\leq t} \, \phi_2'(\pi) = 0$.

If $\pi \models \phi_1 \, U^{\leq t} \, \phi_2$, then by its semantics in PCTL, $\exists j : 0 \leq j \leq t, \pi_j \models \phi_2$, and for all $0 \leq i < j, \pi_i \models \phi_1$, which is equivalent to say, $\exists j : 0 \leq j \leq t$ such that $\phi_2'(\pi_j) = 1$, and for all $0 \leq i < j, \phi_1'(\pi_i) = 1$. The reverse also holds trivially by their definition.

If $\pi \not\models \phi_1 \, U^{\leq t} \, \phi_2$, then by its semantics in PCTL, there does not exist a $j : 0 \leq j \leq t, \pi_j \models \phi_2$, or for some $j : 0 \leq j \leq t, \pi_j \models \phi_2$, but before that, there does not a sequence of $0 \leq i \leq j$ such that $\pi_i \models \phi_1$; this implies either there is no j such that $\pi_j(\phi_2') = 0$, or we have $\pi_j(\phi_2') = 0$ for some $j : 0 \leq j \leq t$, and before that there exists some i, with $\pi_i(\phi_1') = 0$, therefore, we have $\phi_1' \, U_{\times}^{\leq t} \, \phi_2'(\pi) = 0$. If $\phi_1' \, U_{\times}^{\leq t} \, \phi_2'(\pi) = 0$, then this implies either there does not exist a j such that $\pi_j \not\models \phi_2$, or for some $i, \pi_i \not\models \phi_1, i < j$, hence, we have $\pi \not\models \phi_1 \, U^{\leq t} \, \phi_2$.

– $s \models P_{\bowtie v} X\phi$ iff $MX\phi' \bowtie v(s) = 1$; and $s \not\models P_{\bowtie v} X\phi$ iff $MX\phi' \bowtie v(s) = 0$. Let ξ and ξ' denote the set of paths such that $\pi \models X\phi$ and $X\phi'(\pi) = 1$, with respect to starting state s, in PCTL and CTML, respectively. By previous induction, we know $\xi' = \xi$. Then, by PCTL's semantics, we have $s \models P_{\bowtie v} X\phi$ iff $\left(\sum_{\pi \in \xi} \mathbf{P}[\pi_0, \pi_1]\right) \bowtie v$. By CTML's semantics, we have $MX\phi'(s) = \sum_{\pi \in \xi'} X\phi'(\pi) \cdot \mathbf{P}[\pi_0, \pi_1] = \sum_{\pi \in \xi'} \mathbf{P}[\pi_0, \pi_1]$. Therefore, we have $s \models P_{\bowtie v} X\phi$ iff $MX\phi' \bowtie v(s) = 1$, and $s \not\models P_{\bowtie v} X\phi$ iff $MX\phi' \bowtie v(s) = 0$.

– $s \models P_{\bowtie v} \phi_1 U^{\leq t} \phi_2$ iff $M\phi_1' U_{\times}^{\leq t} \phi_2' \bowtie v(s) = 1$; and $s \not\models P_{\bowtie v} \phi_1 U^{\leq t} \phi_2$ iff $M\phi_1' U_{\times}^{\leq t} \phi_2' \bowtie v(s) = 0$. Let ξ and ξ' denote the set of paths such that $\pi \models \phi_1 U^{\leq t} \phi_2$ and $\phi_1' U_{\times}^{\leq t} \phi_2'(\pi) = 1$, with respect to starting state s, in PCTL and CTML, respectively. By previous induction, we know $\xi' = \xi$. Then, by PCTL's semantics, we have $s \models P_{\bowtie v} \phi_1 U^{\leq t} \phi_2$ iff $\left(\sum_{\pi \in \xi} \mathbf{P}[\pi_0, \pi_1] \cdot \cdots \cdot \mathbf{P}[\pi_{t-1}, \pi_t]\right) \bowtie v$. By CTML's semantics, we have $M\phi_1' U_{\times}^{\leq t} \phi_2'(s) = \sum_{\pi \in \xi'} \phi_1' U_{\times}^{\leq t} \phi_2'(\pi) \cdot \mathbf{P}[\pi_0, \pi_1] \cdots \cdots \mathbf{P}[\pi_{t-1}, \pi_t] = \sum_{\pi \in \xi'} \mathbf{P}[\pi_0, \pi_1] \cdots \cdots \mathbf{P}[\pi_{t-1}, \pi_t]$. Therefore, we have $s \models P_{\bowtie v} \phi_1 U^{\leq t} \phi_2$ iff $M\phi_1' U_{\times}^{\leq t} \phi_2' \bowtie v(s) = 1$, and $s \not\models P_{\bowtie v} \phi_1 U^{\leq t} \phi_2$ iff $M\phi_1' U_{\times}^{\leq t} \phi_2' \bowtie v(s) = 0$.

Table 7.1: A Strict Subset of CTML Formulas that Cover PCTL

PCTL	A Subset of CTML Formulas
ϕ, ϕ_1, ϕ_2 (state formula)	ϕ', ϕ'_1, ϕ'_2 (restricted state formula)
$\neg\phi$	$\phi' < one$
$\phi_1 \wedge \phi_2$	$\phi'_1 \cdot \phi'_2$
$\phi_1 \vee \phi_2$	$(\phi'_1 + \phi'_2) > 0$
$X\phi$	$X\phi'$
$\phi_1 U^{\leq t} \phi_2$	$(\phi'_1 U_{\times}^{\leq t} \phi'_2)$
$P_{\bowtie v} X\phi$	$MX\phi' \bowtie v$
$P_{\bowtie v} \phi_1 U^{\leq t} \phi_2$	$(M\phi'_1 U_{\times}^{\leq t} \phi'_2) \bowtie v$

Table 7.1 lists a strict subset of CTML formulas that covers PCTL.

□

Corollary 49. *CTML is strictly more expressive than PCTL.*

Proof. From Theorem 48, we know for each PCTL formula ϕ, there is a CTML equivalent formula ϕ' such that $s \models \phi$ iff $\phi'(s) = 1$; and $s \not\models \phi$ iff $\phi'(s) = 0$. Since CTML has additional operators with no PCTL counterparts such as U_+, V_+, etc., it follows that CTML is strictly more expressive than PCTL. □

CHAPTER 8. ACTION AND STATE BASED FORMALISMS

In this chapter, we describe a new formal language that is syntactically the same, but semantically more powerful than CTML, for supporting multiple actions, and paths featured by a combination of states and actions. The new language is largely inspired by asCSL [5] and CSL$^{\mathrm{TA}}$ [29], for reasoning about both state and action properties over probabilistic systems. Unlike asCSL and CSL$^{\mathrm{TA}}$, however, this language works naturally with the performance measures of *reward* functions [17, 18, 55] that are used to be specified within high level models. In the following, we first give the definition of the language. Then, we present a translation algorithm that converts asCTML to CTML, and a proof for the correctness of the algorithm. After that, we discuss how CTML can be simulated through asCTML as interim results. Finally, we discuss some closely related works.

8.1 Action and State Based Computation Measurement Language

Action and state based computation measurement language, or asCTML for short, is designed to express performance queries based on action and/or state properties over probabilistic systems. The main differences between asCTML and CTML are in their semantics, the interpretation structure, the underlying paths, and the basic set of atomic functions. The following discusses in more detail.

Definition 50 (MAMC structure)**.** *A DTMC with multiple actions, or MAMC for short, is a tuple ($\mathcal{S}$, $\mathcal{ACT}$, γ, δ, $\boldsymbol{\pi}_0$), where*

- *$\mathcal{S}$ is a finite set of states.*
- *$\mathcal{ACT}$ is a finite set of action symbols.*
- *$\gamma : \mathcal{S} \times \mathcal{ACT} \to \mathcal{S}$ specifies the state transitions.*

- $\delta : \mathcal{S} \times \mathcal{ACT} \to [0,1]$ *specifies the transition probabilities, under the constraint that* $\forall s \in \mathcal{S}$, $\delta(s, \mathcal{ACT}) = \sum_{a \in \mathcal{ACT}} \delta(s,a) = 1$.

- $\boldsymbol{\pi}_0 : \mathcal{S} \to [0,1]$ *is an initial probability distribution with* $\sum_{s \in \mathcal{S}} \boldsymbol{\pi}_0[s] = 1$.

Note that by the definition of γ, if $(s,a) = (s,b)$, then $\gamma(s,a) = \gamma(s,b)$ and $t = t'$, where t, t' are the next states that a, b lead to, respectively. That is, given a state s, different next states implies different actions associated with s.

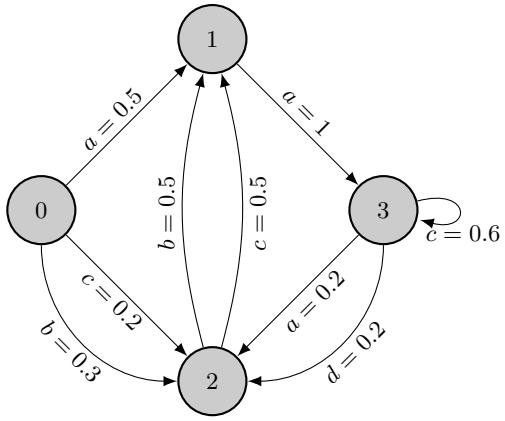


Figure 8.1: An Example of MAMC Structure

Figure 8.1 shows an example of MAMC structure, with $\mathcal{S} = \{0,1,2,3\}$, $\mathcal{ACT} = \{a,b,c,d\}$; at state 0, action a occurs, leading to state 1, with probability 0.5; when action b occurs, leading to state 2, with probability 0.3; when action c occurs, leading to state 2, with probability 0.2; at state 2, actions b and c occur, leading to state 1, with probability 0.5 each, so on and so forth.

Definition 51 (path in MAMC)**.** *Let* $Y = (\mathcal{S}, \mathcal{ACT}, \Gamma, \boldsymbol{\pi}_0)$ *be a MAMC structure, a path in* Y *is an infinite sequence* $\pi = (s_0,a_0),(s_1,a_1),\ldots \in (\mathcal{S} \times \mathcal{ACT})^\omega$*, with* $\pi_i = (s_i,a_i)$*,* $\delta(s_i,a_i) > 0$*,* $s_{i+1} = \gamma(s_i,a_i)$*, and* $i \in \mathbb{N}$.

Given a MAMC structure Y, a *path formula* is a function $\psi : (\mathcal{S} \times \mathcal{ACT})^\omega \to \mathbb{R}^*$, where $\mathbb{R}^*$ denotes nonnegative reals. A *prefix* in an MAMC structure Y is a finite sequence of state, action pairs

$$\mathbf{p} = (s_0,a_0),\ldots,(s_{n-1},a_{n-1}) \in (\mathcal{S} \times \mathcal{ACT})^n,$$

or an infinite sequence $\mathbf{p} = (s_0, a_0), (s_1, a_1) \ldots \in (\mathcal{S} \times \mathcal{ACT})^\omega$, where $\mathbf{p}_i$ is the i^{th} element in $\mathbf{p}$, and $|\mathbf{p}| = n \in \mathbb{N} \cup \{\omega\}$ is the length of the sequence. For a given prefix $\mathbf{p}$, define $\Omega^\omega_\mathbf{p}$ as the set of all infinite length paths that start with prefix $\mathbf{p}$. If $|\mathbf{p}| = n \in \mathbb{N}$, we have

$$\Omega^\omega_\mathbf{p} = (s_0, a_0) \times \ldots \times (s_{n-1}, a_{n-1}) \times \Omega^\omega \tag{8.1}$$

otherwise, if $|\mathbf{p}| = \omega$, then we have $\Omega^\omega_\mathbf{p} = \{\mathbf{p}\}$.

Definition 52 (determines ψ on MAMC)**.** *We say a prefix* $\mathbf{p}$ determines ψ *if, for any paths* $\mathbf{x}, \mathbf{x}' \in \Omega^\omega_\mathbf{p}$, $\psi(\mathbf{x}) = \psi(\mathbf{x}')$*; since all paths must have the same value for* ψ *in this case, we denote this quantity as* $\psi(\mathbf{p})$*. Note that any infinite prefix determines* ψ*.*

Definition 53 (finitely measurable ψ on MAMC)**.** *We say a path formula* ψ *is* finitely measurable *on a MAMC structure* Y *if, for every path* $\mathbf{x} \in \Omega^\omega_\mathbf{p}$ *with* $\psi(\mathbf{x}) > 0$*, either there exists a finite prefix* $\mathbf{p}$ *with* $\mathbf{x} \in \Omega^\omega_\mathbf{p}$ *that determines* ψ*, or the probability measure for path* $\mathbf{x}$ *is zero.*

Like CTML, we wish to define a measure of the expected value of a path formula ψ.

Definition 54 (measure of the expected value of ψ on MAMC)**.** *For any finitely measurable formula* ψ *on* $Y = (\mathcal{S}, \mathcal{ACT}, \gamma, \delta, \boldsymbol{\pi}_0)$*, we define the measure of the expected value* $\mu_\psi : \mathcal{G}_{\pi_0} \to \mathbb{R}^*$ *by*

$$\mu_\psi(\Omega^\omega_\mathbf{p}) = \begin{cases} \psi(\mathbf{p}) \prod_{i=1}^{|\mathbf{p}|-1} \delta(s_i, a_i) & \text{if } \mathbf{p} = ((s_0, a_0), (s_1, a_1), \ldots) \text{ determines } \psi \\ \sum_{(s,a) \in \mathcal{S} \times \mathcal{ACT}} \mu_\psi(\Omega^\omega_{(\mathbf{p},(s,a))}) & \text{otherwise} \end{cases}$$

with $\mu_\psi(\emptyset) = 0$*.*

Note that in this measure definition of the extended model of MAMC, since the starting point is now a pair of state, action, the probability of the initial action associated with (s_0, a_0) is not included, rather, it is treated as part of the initial distribution and captured at the end.

Like CTML, the range of values allowed by asCTML formulas depends on the operator. Unlike CTML, however, asCTML has a new type of basic formula called *state+action* formula.

Definition 55 (state+action formula)**.** *A state+action formula φ formula is defined as a function that maps from a pair of state and action to a nonnegative real value.*

$$\varphi : \mathcal{S} \times \mathcal{ACT} \rightarrow \mathbb{R}^*$$

Definition 56 (restricted state+action formula)**.** *A restricted state+action formula φ_r is defined as a function that maps from a pair of state and action to a real value in interval* $[0, 1]$.

$$\varphi_r : \mathcal{S} \times \mathcal{ACT} \rightarrow [0, 1]$$

In principle, a state+action formula f ranges over $\mathcal{S}$ and $\mathcal{ACT}$, meaning they are defined on every pair of $(s, a) \in \mathcal{S} \times \mathcal{ACT}$; in practice, however, only the pairs with positive value of f are given explicitly. To this end, we assume the following sets of atomic items that are required for the specification of asCTML, but not required to be tied to a model.

- a finite set AF of atomic *state+action* formulas,
- $AR \subseteq AF$ of restricted atomic *state+action* formulas.

For convenience, we also assume $one, zero \in AR$, that evaluate to 1 and 0, respectively, over any element $(s, a) \in (\mathcal{S} \times \mathcal{ACT})$.

8.1.1 asCTML Syntax

Given the definition of atomic *state+action* formula, the syntax of asCTML can be defined the same as CTML, except:

- asCTML operates on state+action formulas, rather than merely state formulas.
- asCTML supports paths of combination of states and actions, whereas CTML supports paths of sequences of states only.
- asCTML distinguishes among multiple actions, whereas CTML does not.

Definition 57 (syntax of asCTML)**.** *Let f be a state+action formula, let r be a restricted state+action formula, and let ψ_r be a restricted path formula defined as $\psi_r : (\mathcal{S} \times \mathcal{ACT})^\omega \to [0,1]$, the syntax of asCTML can be recursively defined as follows:*

$$\begin{aligned}
\varphi &::= f \mid \varphi_r \mid \varphi \odot \varphi \mid M\psi \\
\varphi_r &::= r \mid 1 - \varphi_r \mid \varphi \bowtie \varphi \mid \varphi_r \cdot \varphi_r \mid M\psi_r \\
\psi &::= X\varphi \mid \varphi\, U_{\odot}^{\leq t} \varphi \mid \varphi\, V_{\odot}^{\leq t} \varphi \mid \varphi\, U_{+}\, \varphi \mid \varphi_r\, U_{\times}\, \varphi \mid \varphi_r\, V_{\times}\, \varphi \\
\psi_r &::= X\varphi_r \mid \varphi_r\, U_{\odot}^{\leq t}\, \varphi_r \mid \varphi_r\, V_{\odot}^{\leq t}\, \varphi_r \mid \varphi_r\, U_{\times}\, \varphi_r \mid \varphi_r\, V_{\times}\, \varphi_r
\end{aligned}$$

where $\bowtie \in \{\leq, \geq, <, >\}$, $\odot \in \{+, \times\}$, $f \in AF$, and $f_r \in AR$.

Finally we note that asCTML's top level formula is a *state+action* formula.

8.1.2 Semantics of asCTML

We now give the formal semantics of the operators appearing in the language. Unlike CTL, ACTL, or PCTL, etc., which define a satisfaction relation, asCTML formulas are defined as real-valued functions.

Definition 58 (Semantics of asCTML)**.** *Semantics of asCTML are inductively defined as follows:*

- *If $\varphi = f$, then $\varphi(s,a) = f(s,a)$, $\forall f \in AF$.*
- *If $\varphi = \varphi_1 \cdot \varphi_2$, then $\varphi(s,a) = \varphi_1(s,a) \cdot \varphi_2(s,a)$.*
- *If $\varphi = \varphi_1 + \varphi_2$, then $\varphi(s,a) = \varphi_1(s,a) + \varphi_2(s,a)$.*
- *If $\varphi = \varphi_1 \bowtie \varphi_2$, with $\bowtie \in \{>, \geq, <, \leq\}$ then $\varphi(s,a) = 1$ if $\varphi_1(s,a) \bowtie \varphi_2(s,a)$ holds; $\varphi(s,a) = 0$ otherwise.*
- *If $\varphi = 1 - \varphi_1$, then $\varphi(s,a) = 1 - \varphi_1(s,a)$.*
- *If $\varphi = M\psi$, then $\varphi(s,a) = \mu_\psi(\mathcal{S}^\omega_{s,a})$.*
- *If $\psi = X\varphi$, then $\psi(\pi_0, \pi_1, \ldots) = \varphi(\pi_1)$,*
- *If $\psi = \varphi_1 U_{\odot}^{\leq t} \varphi_2$, then*

– *if* $\exists j \leq t$, $\varphi_2(\pi_j) > 0$, *and* $\forall i < j$, $\varphi_2(\pi_i) = 0$,

$$\psi(\pi_0, \pi_1, \ldots) = \left(\bigodot_{i=0}^{j-1} \varphi_1(\pi_i) \right) \cdot \varphi_2(\pi_j)$$

– *otherwise,* $\psi(\pi_0, \pi_1, \ldots) = 0$.

Also, we have $\varphi_1 U_{\odot}^{\leq \infty} \varphi_2 \equiv \varphi_1 U_{\odot} \varphi_2$.

- *If* $\psi = \varphi_1 V_{\odot}^{\leq t} \varphi_2$, *then*

  – *if* $\exists j \leq t$, $\varphi_2(\pi_j) > 0$, *and* $\forall i < j$, $\varphi_2(\pi_i) = 0$,

  $$\psi(\pi_0, \pi_1, \ldots) = \left(\bigodot_{i=0}^{j-1} \varphi_1(\pi_i) \right) \cdot \varphi_2(\pi_j)$$

  – *otherwise, if* $\forall i \leq t$, $\varphi_2(i) = 0$ *then*

  $$\psi(\pi_0, \pi_1, \ldots) = \bigodot_{i=0}^{t} \varphi_1(\pi_i)$$

  Also, we have $\varphi_1 V_{\times}^{\leq \infty} \varphi_2 \equiv \varphi_1 V_{\times} \varphi_2$.

Note that the finite measurable property of asCTML path formula ψ can be proved exactly the same way as we did for CTML in the previous chapter, so we omit the repetition here.

Finally, we can define the measure of the top-level formula φ on MAMC structure M based on the initial distribution as follow.

Definition 59 (measure of M_φ)**.** *Given a MAMC structure* $M = (\mathcal{S}, \mathcal{ACT}, \gamma, \delta, \boldsymbol{\pi}_0)$*, the total value of a* state+action *formula* φ *on* M *is defined by*

$$M_\varphi = \sum_{s \in \mathcal{S}} \left(\sum_{a \in \mathcal{ACT}} \varphi(s, a) \cdot \delta(s, a) \right) \cdot \boldsymbol{\pi}_0(s). \tag{8.2}$$

8.2 Conversion to CTML

In this section, we describe a formal translation method that takes a MAMC structure and a set of asCTML formula and produces a DTMC structure and a corresponding set CTML formula. Then we illustrate this mapping on an example. Finally, we prove that the translated DTMC + CTML formula always gives the same value as the MAMC + asCTML formula.

MAMC+asCTML to DTMC+CTML Translation Algorithm: Given a MAMC structure $M = (\mathcal{S}, \mathcal{ACT}, \gamma, \delta, \boldsymbol{\pi}_0)$, a corresponding DTMC $D = (\mathcal{S}', \mathbf{P}, \boldsymbol{\pi}'_0)$ can be built by the following:

- $\mathcal{S}' = \{(sa) | (sa) \in \mathcal{S} \times \mathcal{ACT}, \delta(s,a) > 0\}$.
- For all $(sa), (s'a') \in \mathcal{S}'$,

$$\mathbf{P}[(sa),(s'a')] = \begin{cases} \delta(s',a') & \text{If } \ \gamma(s,a) = s', \\ 0 & \text{otherwise.} \end{cases}$$

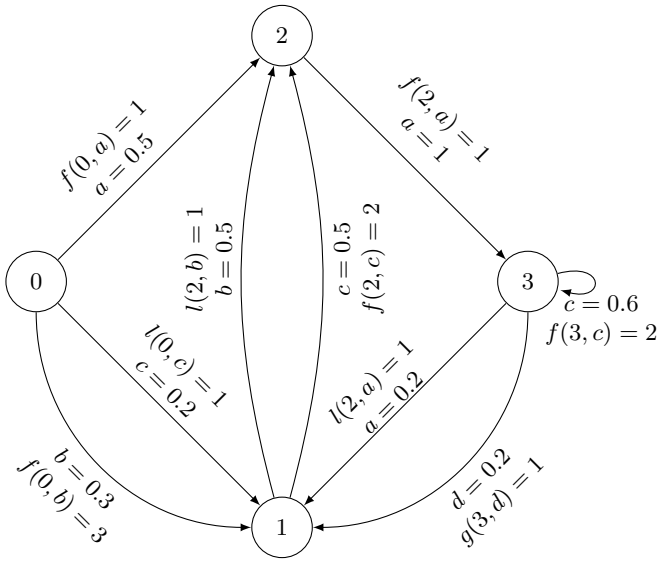
- $\boldsymbol{\pi}'_0[(sa)] = \boldsymbol{\pi}_0[s] \cdot \delta(s,a)$.
- The translation for the set of atomic state+action formulas $\mathcal{AF}$ is simply that for each $f \in \mathcal{AF}$ on M, we have $f' \in \mathcal{AF}'$ on D, with $f(s,a) = f'(sa)$, for all $(sa) \in \mathcal{S}'$.

  Note that since the two sets $\mathcal{AF}$ and $\mathcal{AF}'$ are exactly the same except one is defined on M with the parameter (s,a) being a pair of state and action and the other is defined on D with parameter (sa) being a state, for the following discussion, we use notations such as f^M and f^D, respectively, to refer to the same formulas on different model structures.

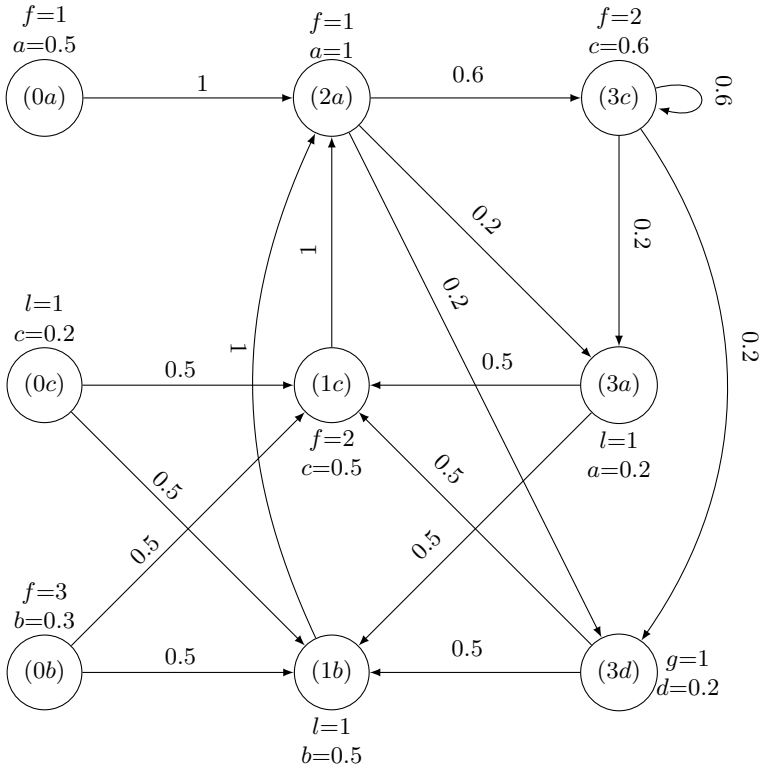
- Then, for any asCTML formula ϕ on M, denoted by ϕ^M, is translated to the same formula ϕ on D, denoted by ϕ^D, such that for each atomic state+action formula f^M in ϕ^M, it is replaced by the corresponding f^D in ϕ^D.

Figure 8.2 shows an example of translation from MAMC to DTMC, and from a set of asCTML atomic+action formulas to CTML atomic state formulas.

Theorem 60. *Let ϕ^M be an asCTML formula on MAMC structure M, and D be the translated DTMC structure, then $\phi^M(s,a)$ evaluate the same as $\phi^D(sa)$ on D, where ϕ^D is the translation of ϕ^M.*



(a) A MAMC structure and asCTML formulas before translation



(b) The Translated DTMC + CTML formulas

Figure 8.2: An Example Translation from MAMC to DTMC

Proof. According to the translation method, there is a one to one mapping between state+action pair (s, a) on M and state (sa) on D, and each atomic state+action formula f on M is mapped to an atomic state formula f on D. As such, $f^M = f^D$ holds trivially. By structural induction, assume that $\phi^M = \phi^D$, $\phi_1^M = \phi_1^D$, and $\phi_2^M = \phi_2^D$. Then,

- $(\phi_1 \cdot \phi_2)^M = (\phi_1 \cdot \phi_2)^D$, $(\phi_1 + \phi_2)^M = (\phi_1 + \phi_2)^D$, $(\phi_1 \bowtie \phi_2)^M = (\phi_1 \bowtie \phi_2)^D$, and $(1 - \phi)^M = (1 - \phi)^D$ hold trivially, by the assumption.

  Note that for succinctness, when the context is clear (e.g., when the parameter is given as a state such as (sa), then it is a CTML formula; if the parameter is given as a pair of state and action such as (s, a), then it is an asCTML formula), we drop the the superscript D and M, respectively.

  Also, for the following proof of $M\psi$, we utilize the fact that there exists a one to one mapping between each state, action pair (s, a) on MAMC and the translated state (sa) on DTMC. Then, each element (s_i, a_i) in a given MAMC path $\pi^M = (s, a), (s_1, a_1), \cdots$ matches the state $(s_i a_i)$ in the corresponding path $\pi^D = sa, s_1 a_1, \cdots$ of the translated DTMC.

- $(MX\phi)^M = (MX\phi)^D$. By asCTML semantics on $MX\phi$, for all paths: $\pi = (s, a), (s_1, a_1), \cdots$, we have:

$$
\begin{aligned}
(MX\phi)^M(s, a) &= \sum_{(s_1, a_1) \in \{\gamma(s,a)\} \times \mathcal{ACT}} \phi(s_1, a_1)\delta(s_1, a_1) \\
&\qquad \text{By the assumption, and translations given } s_1 = \gamma(s, a), \\
&= \sum_{(s_1 a_1) \in \mathcal{S}'} \phi(s_1 a_1) \cdot \mathbf{P}[sa,\ s_1 a_1] \\
&= (MX\phi)^D(sa).
\end{aligned}
$$

- $(M\phi_1 U_{\odot}^{\leq t} \phi_2)^M = (M\phi_1 U_{\odot}^{\leq t} \phi_2)^D$. By asCTML semantics on $M\phi_1 U_{\odot}^{\leq t} \phi_2$, for all paths of $\pi = (s, a), (s_1, a_1), \cdots$, with $(s, a) = (s_0, a_0)$, if $\exists j : 0 \leq j \leq t$, s.t. $\phi_2(s_j, a_j) > 0$ and for all

$0 \leq i \leq j$, $\phi_2(s_i, a_i) = 0$, we have:

$$
\begin{aligned}
(M\phi_1 U_{\odot}^{\leq t} \phi_2)^M(s,a) &= \sum_{j=0}^{t} \sum_{\substack{((s_0,a_0),\ldots,(s_j,a_j))\in(\mathcal{S}\times\mathcal{ACT})^j, \\ s_{i+1}=\gamma(s_i,a_i)}} \left(\bigodot_{i=0}^{j-1} \phi_1(s_i,a_i)\right) \cdot \phi_2(s_j,a_j) \cdot \prod_{i=1}^{j} \delta(s_i,a_i) \\
&\qquad \text{By the assumption, and translations given } s_i = \gamma(s_{i-1}, a_{i-1}), \\
&= \sum_{j=0}^{t} \sum_{(s_0a_0,\ldots,s_ja_j)\in\mathcal{S}'^j} \left(\bigodot_{i=0}^{j-1} \phi_1(s_ia_i)\right) \cdot \phi_2(s_ja_j) \cdot \prod_{i=1}^{j} \mathbf{P}[s_{i-1}a_{i-1},\ s_ia_i] \\
&= (M\phi_1 U_{\odot}^{\leq t} \phi_2)^D(sa).
\end{aligned}
$$

- $(M\phi_1 V_{\odot}^{\leq t} \phi_2)^M(s,a) = (M\phi_1 V_{\odot}^{\leq t} \phi_2)^D(sa)$. According to the asCTML and CTML semantics of $M\phi_1 V_{\odot}^{\leq t} \phi_2$, there are two cases for this formula. One case is that if $\exists j : 0 \leq j \leq t$, s.t. $\phi_2(s_j, a_j) > 0$ and for all $0 \leq i \leq j$, $\phi_2(s_i, a_i) = 0$. In this case, we have just established that $(M\phi_1 U_{\odot}^{\leq t} \phi_2)^M = (M\phi_1 U_{\odot}^{\leq t} \phi_2)^D$. The second case is when $\forall i : i \leq t, \phi_2(s_i, a_i) = 0$. In this case, we have: we have:

$$
\begin{aligned}
(M\phi_1 V_{\odot}^{\leq t} \phi_2)^M(s,a) &= \sum_{\substack{((s_0,a_0),\ldots,(s_t,a_t))\in(\mathcal{S}\times\mathcal{ACT})^t \\ s_{i+1}=\gamma(s_i,a_i)}} \left(\bigodot_{i=0}^{t} \phi_1(s_i,a_i)\right) \cdot \prod_{i=1}^{t} \delta(s_i,a_i) \\
&\qquad \text{By the assumption, and translations given } s_i = \gamma(s_{i-1}, a_{i-1}), \\
&= \sum_{(s_0a_0,\ldots,s_ta_t)\in\mathcal{S}'^t} \left(\bigodot_{i=0}^{t} \phi_1(s_ia_i)\right) \cdot \prod_{i=1}^{t} \mathbf{P}[s_{i-1}a_{i-1},\ s_ia_i] \\
&= (M\phi_1 V_{\odot}^{\leq t} \phi_2)^D(sa).
\end{aligned}
$$

  Therefore, we have $(M\phi_1 V_{\odot}^{\leq t} \phi_2)^M = (M\phi_1 V_{\odot}^{\leq t} \phi_2)^D$.

- Finally, according to the CTML definition 41 for the final value of ϕ on D, denoted by D_ϕ and asCTML definition 59 for the final value of ϕ on M, denoted by M_ϕ, we have:

$$
\begin{aligned}
M_\phi &= \sum_{s\in\mathcal{S}} \left(\sum_{a\in\mathcal{ACT}} \phi(s,a) \cdot \delta(s,a) \right) \cdot \boldsymbol{\pi}_0(s) \\
&= \sum_{(sa)\in\mathcal{S}'} \phi(sa) \cdot \boldsymbol{\pi}'_0(sa) \quad \text{by the translation algorithm} \\
&= D_\phi
\end{aligned}
$$

□

8.3 asCTML vs. CTML

In section 8.1, we have defined asCTML and CTML in such a way that they share the same syntax, but having different semantics, with asCTML operating on state+action formulas whereas CTML operating on state formulas. Given the similarities between asCTML and CTML, we are interested in exploring the relation between the two languages, in particular, we are interested in finding out whether an asCTML formula can be used to express a corresponding CTML formula directly. Note that for the following discussion, if a formula ϕ has a single state parameter, then it is a CTML formula; if its parameter is a pair of state and action, then it is an asCTML formula.

By their definition, the top-level asCTML formula ϕ returns $\sum_s(\sum_a \phi(s,a)\delta(s,a))\boldsymbol{\pi}_0[s]$ and the corresponding top-level CTML formula ϕ' returns $\sum_s \phi'(s)\boldsymbol{\pi}_0[s]$. These will be equal if $\sum_a \phi(s,a)\delta(s,a) = \phi'(s)$, for all s with $\boldsymbol{\pi}_0[s] > 0$. Further, we can only guarantee $\sum_a \phi(s,a)\delta(s,a) = \phi'(s)$ if all the operands f' in ϕ' such that $f'(s) = f(s,a)$. over all actions.

Take the Figure 8.3 for example. Suppose the initial distribution is $\boldsymbol{\pi}_0[0] = 1$. Let the CTML and asCTML atomic formulas g', h', and g, h be defined as follows, respectively.

- $g'(2) = g(2,a) = 1$, and
- $h'(3) = h(3,b) = 1$.

We are interested in finding out whether $\sum_{a\in\mathcal{ACT}}(M(MXg)U_+\, h)(s,a)\delta(s,a) = (M(MXg')U_+\, h')(s)$. For convenience, let $\phi'_1 = MXg'$, and $\phi' = M(MXg')U_+\, h'$. By CTML's semantics, we have $\phi'_1(1) = MXg'(1) = 1\cdot\alpha + 0 = \alpha$, with value 0 for all other states. By asCTML's semantics, we have $\phi_1(1,a) = MXg(1,a) = g(2,a)\delta(2,a) = 1$, with all others having value 0 for ϕ_1, then $\sum_{a\in\mathcal{ACT}}\phi_1(1,a)\delta(1,a) = \alpha$. In this case, we have $\phi'_1(1) = \sum_{a\in\mathcal{ACT}}\phi_1(1,a)\delta(1,a)$, because we have the operand $g'(s) = g(s,a)$ over all actions for formula MXg.

Now, continuing this, for CTML formula ϕ, we have $\phi'(0) = (M\phi_1U_+\, h)(0) = (0+\alpha)1(1-\alpha) = \alpha(1-\alpha)$, because we only have one path $(0,1,3,3,\ldots)$ that leads to $h > 0$. For the corresponding asCTML formula ϕ, however, we obtain $\phi(0,a) = (M\phi_1U_+\, h)(0,a) = 0$, and $\sum_{a\in\mathcal{ACT}}\phi(0,a)\delta(0,a) =$

0. Thus, we have $\sum_{a\in\mathcal{ACT}}\phi(0,a)\delta(0,a) = 0 \neq \phi'(0) = \alpha(1-\alpha)$, because we have the operand $MXg'(1) \neq MXg(1,a)$ for formula $M\phi_1 U_+ h$. This example shows that once we have an operand $f(s) \neq f(s,a)$ for any $a \in \mathcal{ACT}$, we cannot guarantee that $\phi(s)$ is equal to $\sum_{a\in\mathcal{ACT}}\phi(s,a)\delta(s,a)$.

In other words, though strictly speaking CTML cannot be expressed by asCTML "purely", it can be simulated by asCTML with proper steps. The overall idea about the simulation is that, for each case of CTML formula (i.e., it contains one valid CTML operator in the set of $\{\bowtie, \odot, 1-, MX, MU, MV\}$), we set the value of atomic formulas such that $f(s,a) = f(s)$, then compute the corresponding asCTML formula, then sum over the actions of the computed asCTML formula. Algorithm 8.1 gives details about the simulation, for which we assume a parse tree has been built for a given CTML formula ϕ'.

Consider the example as shown in Figure 8.3 again. Let $\phi' = M(MXg')U_+ h'$. Initially we call $simulate(\phi', M)$. Then, it goes to the case of $M\phi_1' U_+^{\leq t}\phi_2'$, with $\phi_1' = MXg'$, so it calls $simulate(\phi_1', M)$. Then it goes to the case of $MX\phi'$. Again, it calls $simulate(g', M)$, which returns $\sum_{a\in\mathcal{ACT}} g(s,a)\delta(s,a)$. Then $\phi_1'(s) = MXg'(s) = \sum_{a\in\mathcal{ACT}}\phi_1(s,a)\delta(s,a)$ gives $\phi_1'(1) = MXg'(1) = \alpha$ and 0 for all others. Then it sets $\phi_1(1,a) = \phi_1(1,b) = \phi_1'(1) = \alpha$ and $h(s,a) = h'(s)$. Finally, we have $\phi(s) = M\phi_1' U_+ h'(s) = \sum_{a\in\mathcal{ACT}}\phi(s,a)\delta(s,a) = \sum_{a\in\mathcal{ACT}}(M\phi_1 U_+ h)(s,a)\delta(s,a) = \alpha(1-\alpha)$. The following theorem generalizes the idea of the simulation.

Theorem 61. *Let $D = (\mathcal{S}, \mathbf{P}, \boldsymbol{\pi}_0)$ be a DTMC structure obtained from an MAMC structure $M = (\mathcal{S}, \mathcal{ACT}, \gamma, \delta, \boldsymbol{\pi}_0)$, with $\mathbf{P}[s,s'] = \sum_{a:s'=\gamma(s,a)}\delta(s,a)$. Let ϕ' be a CTML formula on D. Then, algorithm 8.1 computes the correct value for ϕ'.*

Proof.

- $f'(s) = f'(s)\sum_{a\in\mathcal{ACT}}\delta(s,a) = \sum_{a\in\mathcal{ACT}} f'(s)\delta(s,a) = \sum_a f(s,a)\delta(s,a)$, since $\sum_a \delta(s,a) = 1$.

- Given $\phi_1'(s) = \phi_1(s,a)$ and $\phi_2'(s) = \phi_2(s,a)$, for all $a \in \mathcal{ACT}$. We have: $(\phi_1' \odot \phi_2')(s) = \phi_1'(s) \odot \phi_2'(s) = \phi_1(s,a) \odot \phi_2(s,a) = (\phi_1 \odot \phi_2)(s,a) = (\phi_1 \odot \phi_2)(s,a)\sum_{a\in\mathcal{ACT}}\delta(s,a) = \sum_{a\in\mathcal{ACT}}(\phi_1 \odot \phi_2)(s,a)\delta(s,a)$.

- $(\phi_1' \bowtie \phi_2')(s) = 1$ iff $\phi_1'(s) \bowtie \phi_2'(s)$ holds, and 0 otherwise. Also, we have $(\phi_1 \bowtie \phi_2)(s,a) = 1$ iff $\phi_1(s,a) \bowtie \phi_2(s,a)$ holds, and 0 otherwise. Since $\phi_1'(s) = \phi_1(s,a)$ and $\phi_2'(s) = \phi_2(s,a)$, for

Algorithm 8.1 $simulate(\phi',\ M)$

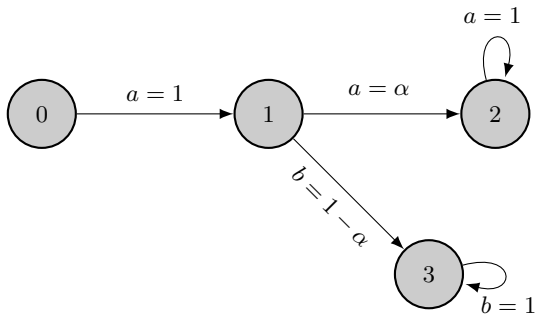
Input: ϕ', M
Output: vector $\boldsymbol{\pi}$ such that $\boldsymbol{\pi}[s] = \phi'(s)$

switch ϕ' **do**
  case f'
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = f'(s)$;
    $f = setOperand(\boldsymbol{\pi}, M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} f(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;
  case $\phi'_1 \odot \phi'_2$
    $\phi_1 = setOperand(simulate(\phi'_1, M),\ M)$;
    $\phi_2 = setOperand(simulate(\phi'_2, M),\ M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} (\phi_1 \odot \phi_2)(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;
  case $\phi'_1 \bowtie \phi'_2$
    $\phi_1 = setOperand(simulate(\phi'_1, M),\ M)$;
    $\phi_2 = setOperand(simulate(\phi'_2, M),\ M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} (\phi_1 \bowtie \phi_2)(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;
  case $1 - \phi'$
    $\phi = setOperand(simulate(\phi', M),\ M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} (1 - \phi)(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;
  case $M\mathrm{X}\phi'$
    $\phi = setOperand(simulate(\phi', M),\ M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} (MX\phi)(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;
  case $M\phi'_1 U^{\leq t}_{\odot} \phi'_2$
    $\phi_1 = setOperand(simulate(\phi'_1, M),\ M)$;
    $\phi_2 = setOperand(simulate(\phi'_2, M),\ M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} (M\phi_1 U^{\leq t}_{\odot} \phi_2)(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;
  case $M\phi'_1 V^{\leq t}_{\odot} \phi'_2$
    $\phi_1 = setOperand(simulate(\phi'_1, M),\ M)$;
    $\phi_2 = setOperand(simulate(\phi'_2, M),\ M)$;
    for $\forall s \in \mathcal{S}$ do
      $\boldsymbol{\pi}[s] = \sum_{a \in \mathcal{ACT}} (M\phi_1 V^{\leq t}_{\odot} \phi_2)(s,a)\delta(s,a)$;
    return $\boldsymbol{\pi}$;

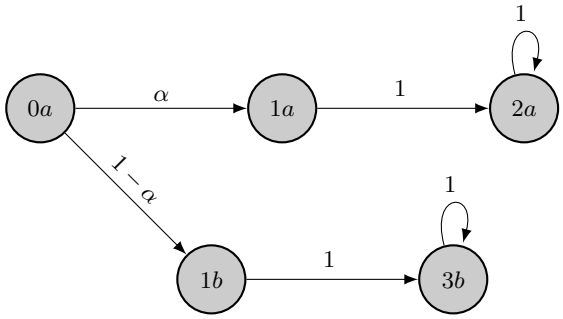
Algorithm 8.2 $setOperand(\boldsymbol{\pi},\ M)$

Input: **vector** $\boldsymbol{\pi}$, M
Output: asCTML atomic formula f such that $f(s,a) = \boldsymbol{\pi}[s]$

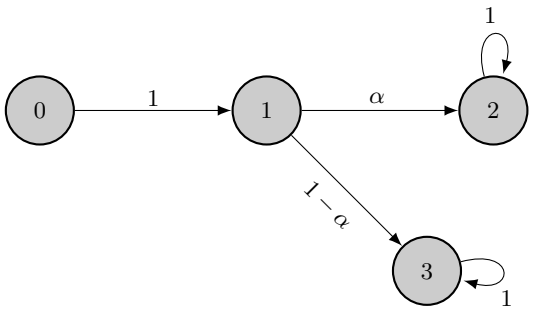
```
for ∀s ∈ S do
    for ∀a ∈ ACT do
        f(s,a) = π[s];
return f;
```



(a) A MAMC Example for asCTML



(b) The Translated DTMC for asCTML



(c) The Corresponding DTMC for CTML

Figure 8.3: A Example for asCTML vs. CTML

any $a \in \mathcal{ACT}$, we have $(\phi'_1 \bowtie \phi'_2)(s) = (\phi_1 \bowtie \phi_2)(s,a) = (\phi_1 \bowtie \phi_2)(s,a) \sum_{a \in \mathcal{ACT}} \delta(s,a) = \sum_{a \in \mathcal{ACT}} (\phi_1 \bowtie \phi_2)(s,a)\delta(s,a)$.

- $(1-\phi')(s) = one(s) - \phi'(s) = one(s,a) - \phi(s,a) = (1-\phi)(s,a) = (1-\phi)(s,a) \sum_{a \in \mathcal{ACT}} \delta(s,a) = \sum_{a \in \mathcal{ACT}} (1-\phi)(s,a)\delta(s,a)$.

- $(MX\phi')(s) = \sum_{a \in \mathcal{ACT}} (MX\phi)(s,a)\delta(s,a)$. By the semantics of CTML formula $MX\phi'$ on D, given all paths $\pi = (s, s_1, \ldots)$, we have:

$$
\begin{aligned}
(MX\phi')(s) &= \sum_{s_1 \in \mathcal{S}} \phi'(s_1)\mathbf{P}[s,s_1] \\
&= \sum_{s_1 \in \mathcal{S}} \phi'(s_1) \sum_{a:\, s_1 = \gamma(s,a)} \delta(s,a) \qquad \because \mathbf{P}[s,s_1] = \textstyle\sum_{a:\, s_1=\gamma(s,a)} \delta(s,a) \\
&= \sum_{s_1 \in \mathcal{S}} \phi'(s_1) \sum_{a_1 \in \mathcal{ACT}} \delta(s_1,a_1) \sum_{a:\, s_1 = \gamma(s,a)} \delta(s,a) \\
&= \sum_{s_1 \in \mathcal{S}} \sum_{a_1 \in \mathcal{ACT}} \phi'(s_1)\delta(s_1,a_1) \sum_{a:\, s_1 = \gamma(s,a)} \delta(s,a) \\
&= \sum_{(s_1,a_1) \in \mathcal{S} \times \mathcal{ACT}} \phi(s_1,a_1)\delta(s_1,a_1) \sum_{a:\, s_1 = \gamma(s,a)} \delta(s,a) \qquad \because \phi'(s_1) = \phi(s_1,a_1) \\
&= \sum_{a \in \mathcal{ACT}} \delta(s,a) \sum_{(s_1,a_1) \in \{\gamma(s,a)\} \times \mathcal{ACT}} \phi(s_1,a_1)\delta(s_1,a_1) \\
&= \sum_{a} (MX\phi)(s,a)\, \delta(s,a) \quad \text{by the definition of } MX\phi
\end{aligned}
$$

- $(M\phi'_1 U_{\odot}^{\leq t} \phi'_2)(s) = \sum_a (M\phi_1 U_{\odot}^{\leq t} \phi_2)(s,a)\, \delta(s,a)$.

  By the semantics of CTML formula $M\phi'_1 U_{\odot}^{\leq t} \phi'_2$ on D, given all paths $\pi = (s, s_1, \cdots)$, with

$s = s_0$, if $\exists j : 0 \leq j \leq t$, s.t. $\phi_2(s_j) > 0$ and for all $0 \leq i \leq j$, $\phi_2(s_i) = 0$, we have:

$$
\begin{aligned}
(M\phi_1' U_{\odot}^{\leq t} \phi_2')(s) &= \sum_{j=0}^{t} \sum_{(s_0,\ldots,s_j)\in\mathcal{S}^j} \left(\bigodot_{i=0}^{j-1} \phi_1'(s_i) \right) \phi_2'(s_j) \prod_{i=0}^{j} \mathbf{P}[s_i, s_{i+1}] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \because \mathbf{P}[s_i,s_{i+1}] = \sum_{a_i:\, s_{i+1}=\gamma(s_i,a_i)} \delta(s_i,a_i) \\
&= \sum_{j=0}^{t} \sum_{(s_0,\ldots,s_j)\in\mathcal{S}^j} \left(\bigodot_{i=0}^{j-1} \phi_1'(s_i) \right) \phi_2'(s_j) \prod_{i=0}^{j} \sum_{a_i:\, s_{i+1}=\gamma(s_i,a_i)} \delta(s_i, a_i) \\
&= \sum_{j=0}^{t} \sum_{\substack{((s_0,a_0),\ldots,(s_j,a_j))\in(\mathcal{S}\times\mathcal{ACT})^j \\ s_{i+1}=\gamma(s_i,a_i)}} \left(\bigodot_{i=0}^{j-1} \phi_1(s_i, a_i) \right) \phi_2(s_j, a_j) \prod_{i=0}^{j} \delta(s_i, a_i) \\
&= \sum_{a\in\mathcal{ACT}} \delta(s,a) \sum_{j=0}^{t} \sum_{\substack{((s_0,a_0),\ldots,(s_j,a_j))\in(\mathcal{S}\times\mathcal{ACT})^j \\ s_{i+1}=\gamma(s_i,a_i)}} \left(\bigodot_{i=0}^{j-1} \phi_1(s_i, a_i) \right) \phi_2(s_j, a_j) \\
&\quad \cdot \prod_{i=1}^{j} \delta(s_i, a_i) \\
&= \sum_{a\in\mathcal{ACT}} \delta(s,a)(M\phi_1 U_{\odot}^{\leq t} \phi_2)(s,a) \qquad \text{by the definition of } MU
\end{aligned}
$$

Note that in the second equation for the proof of $M\phi_1' U_{\odot}^{\leq t} \phi_2'$, the term $\prod_{i=0}^{j} \sum_{a_i:\, s_{i+1}=\gamma(s_i,a_i)} \delta(s_i, a_i)$ is equal to $\left(\sum_{a_0:\, s_1=\gamma(s_0,a_0)} \delta(s_0, a_0)\right) \cdot \cdots \cdot \left(\sum_{a_j:\, s_{j+1}=\gamma(s_j,a_j)} \delta(s_j, a_j)\right)$, which is essentially the cross product over $\{(s_i, a_i)\}$. So we can eliminate the summation term by replacing the paths of $(s_0, \ldots, s_j)$ with $((s_0, a_0), \ldots, (s_j, a_j))$. Also, in the 4th equation, we simply extract the starting action out, and make the term $\prod_{i=0}^{j} \delta(s_i, a_i)$ start from $i = 1$ rather than $i = 0$, then we derived the asCTML semantics for the corresponding formula.

- $(M\phi_1' V_{\odot}^{\leq t} \phi_2')(s) = \sum_{a\in\mathcal{ACT}} M\phi_1 V_{\odot}^{\leq t} \phi_2(s,a) \cdot \delta(s,a)$. According to the semantics of CTML, there are two cases for the formula of $M\phi_1' V_{\odot}^{\leq t} \phi_2'$, One case is that if $\exists j : 0 \leq j \leq t$, s.t. $\phi_2'(s_j) > 0$ and for all $0 \leq i \leq j$, $\phi_2'(s_i) = 0$. In this case, we have just established that $M\phi_1' U_{\odot}^{\leq t} \phi_2'(s) = \sum_{a\in\mathcal{ACT}} M\phi_1 U_{\odot}^{\leq t} \phi_2(s,a) \cdot \delta(s,a)$. The second case is when $\forall i : i \leq$

$t, \phi_2'(s_i) = 0$. In this case, by CTML's semantics, we have:

$$
\begin{aligned}
(M\phi_1' V_{\odot}^{\leq t} \phi_2')(s) &= \sum_{(s_0,\ldots,s_t)\in\mathcal{S}^t} \bigodot_{i=0}^{t} \phi_1'(s_i) \prod_{i=1}^{t} \mathbf{P}[s_{i-1}, s_i] \\
&\qquad\qquad \because \mathbf{P}[s_i,s_{i+1}] = \sum_{a_i:\, s_{i+1}=\gamma(s_i,a_i)} \delta(s_i,a_i) \\
&= \sum_{(s_0,\ldots,s_t)\in\mathcal{S}^t} \bigodot_{i=0}^{t} \phi_1'(s_i) \prod_{i=0}^{t} \sum_{a_i:\, s_{i+1}=\gamma(s_i,a_i)} \delta(s_i,a_i) \\
&= \sum_{\substack{((s_0,a_0),\ldots,(s_t,a_t))\in(\mathcal{S}\times\mathcal{ACT})^t \\ s_{i+1}=\gamma(s_i,a_i)}} \bigodot_{i=0}^{t} \phi_1(s_i,a_i) \prod_{i=0}^{t} \delta(s_i,a_i) \\
&= \sum_{a\in\mathcal{ACT}} \delta(s,a) \sum_{\substack{((s_0,a_0),\ldots,(s_t,a_t))\in(\mathcal{S}\times\mathcal{ACT})^t \\ s_{i+1}=\gamma(s_i,a_i)}} \bigodot_{i=0}^{t} \phi_1(s_i,a_i) \prod_{i=1}^{t} \delta(s_i,a_i) \\
&= \sum_{a\in\mathcal{ACT}} \delta(s,a)(M\phi_1 V_{\odot}^{\leq t} \phi_2)(s,a) \qquad \text{by the definition of } MV
\end{aligned}
$$

□

8.4 Other Related Work

There is a body of past work on applying formal methods to the analysis of stochastic systems; some of these (namely, [3, 6, 25, 28]) have been briefly discussed already in chapters 1 and 6. In this section, we look into related works that are concerned with actions, and works that are concerned with rewards values.

ACTL [32] and asCSL [5] are related to asCTML in the sense that their path formulas can operate on paths featured by sequences of states and actions. Specifically, ACTL has a primitive set of action formulas similar to that asCTML and its translation process is also analogous to that of asCTML. However, unlike asCTML, ACTL's action formulas cannot be nested. In fact, as it is proven in the paper [54], ACTL has the same expressive power as CTL, hence asCTML is strictly more expressive than ACTL since asCTML covers PCTL which is strictly more expressive than CTL.

asCSL is an extension of CSL [6] and aCSL [36] that work with continuous-time Markov chains, but its path formulas are defined as regular expressions over actions and states. Similar to asCTML,

asCSL allows multiple actions to be associated each transition. Unlike asCTML, however, asCSL does not consider real-valued rewards; in fact, asCTML and asCSL are mostly incomparable.

PRCTL [2], which is very similar to [7] except the latter works with continuous-time Markov chains, and the recent work [41] are more closely related to CTML, rather than asCTML, since they do not have action formulas, but are the extensions of PCTL with rewards functions. Both [2] and [41] present an expected accumulated reward operator that is similar to CTML's bounded weak until, with addition. However, the CTML until and weak until operators with addition take two operands, whereas the cumulative operator defined in [2] takes one operand ϕ that is accumulated over states that satisfy ϕ, and the cumulative operator in [41] accumulates a reward variable that is given by the reward structure. Except for rewards defined on edges which are allowed in [41] but not in CTML, these cumulative operators can be expressed using CTML's bounded weak until operator by using *zero* for the right operand and adjusting the time to be $t-1$. [41] also presents a reachability operator R that is similar to CTML's unbounded until, with addition, except that R takes a single operand, similar to CTL's F operator. There are two main differences between R and CTML's unbounded until operator using *zero* for the left operand: first, R requires that the destination states are eventually reached with probability one, while U does not; second, CTML can use real values to distinguish the destination states, while R cannot.

Another work we would like to mention is *performance trees* [61], as they can describe results of various types (real–valued or otherwise, including distributions). Unlike CTML or asCTML, however, performance trees is a more general framework or interface that utilizes the existing performance evaluation algorithms (such as passage time distributions [34]) as well as some of the existing model checking algorithms for the expression of both logic and real–valued measures.

Additionally, there is work that computes the *probabilistic reachability* and *expected reachability* for Markov decision processes (MDPs) or its variants. More specifically, [23] handles a probabilistic structure where the duration time is either 0 or 1 between state to state transitions. [46] extended the idea by allowing the duration time to be an arbitrary natural number between state to state transitions. [24] and [45] incorporate the real value between state to state transitions for computing the expected reachability, which is then treated as the stochastic shortest path problem [10] for MDPs.

CHAPTER 9. SOFTWARE TOOL

In this chapter, we discuss a prototype software tool, developed for the evaluation of CTML queries. The CTML software tool serves as the backend engine for performing dependability and performance analysis. When the input models are small, they can be hand crafted and directly fed into the tool for automatic analysis. But typically, CTML tool works in conjunction with a front-end engine for automatic generation of probabilistic models and rewards functions, particularly when the input models are large, which is often the case in practice. For asCTML queries, we add an additional translator that translates from MAMC to DTMC and asCTML to CTML.

In the following, Section 9.1 discusses CTML software design. Section 9.2 discusses basic techniques used in the CTML software implementation. Section 9.3 discusses asCTML translator. Section 9.4 discusses overall software test.

9.1 Software Design

As like any other software, CTML tool consists of inputs, outputs, and a control unit which is called CTML engine. The inputs consist of three parts. The first part is a probabilistic model. The second part is a set of rewards functions (a.k.a. atomic functions). The third part is the performance/dependability specifications that are written in CTML. Figure 9.2 shows an input format example for the model shown in Figure 5.1. The input starts with the line `DTMC` to indicate that it is a DTMC structure. It is then followed by the number of states, the initial distribution, the number of edges, then followed by a sequence of edges, ordered by the row number, each of which has a nonzero probability value; the line of `END` indicates the end of model structure. Note that the probability for each edge is normalized in the end before the measure is processed. It is normalized in such a way that we calculate the probability of $i \rightarrow j$ by taking the weight of that edge divided by the total weight of all outgoing edges for state i.

This is convenient, because in some cases the original edge weight may not necessary be a probability, rather, it is some positive real value. Following the model structure is the atomic function specifications in the format of *state : weight* if the state has nonzero weight value. The third part of input is the measure specification. It starts with the reserved word `MEASURE`, followed by a sequence of concrete measures. For easier processing, in this prototype implementation, each measure is assume to be well parenthesized for each CTML operator. These inputs are then fed into CTML engine for automatic processing, and it outputs a set of real values corresponding to each query. Figure 9.1 illustrates a prototype design of the CTML software tool.

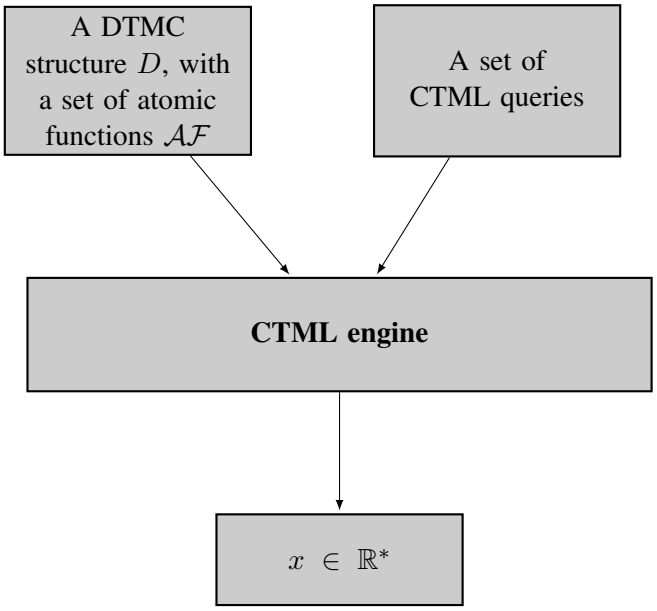


Figure 9.1: CTML Software Structure

The main component of the whole software tool is the CTML engine. It hosts all the algorithms for each CTML formula as we discussed in the previous chapter. The design of CTML engine takes advantage of polymorphism type of object-oriented programing language of Java such that each specific algorithm implements the same interface method called “compute()”. When a concrete operator such as *until-plus*, *until-multiply*, etc. is read, a computation object for the corresponding operator, say until-plus-object, is then instantiated, and the compute method associated with the until-plus-object operator is invoked. Figure 9.3 shows an idea of the polymorphic design feature. Such a design consideration is

```
        DTMC
        STATES 4
        INIT
          0 : 1.0
        ARCS   5
          0 : 1 : 1.0

          1 : 1 : 0.2
          1 : 2 : 0.3
          1 : 3 : 0.5

          2 : 2 : 1.0
          3 : 3 : 1.0
        END

        idle
          0 : 1.0
        end_idle

        trying
          1 : 2.0
        end_trying

        succ
          2 : 1.0
        end_succ

        fail
          3 : 1.0
        end_fail

MEASURE
    (one until_plus succ)
    (one until_mult(trying mult(trying until_mult succ)))
```

Figure 9.2: An Example of Input Model Format

mainly for flexibility in the sense that if there will be any extensions in the future, then new algorithms can be added into the tool by its own without largely affecting the existing design and implementation.

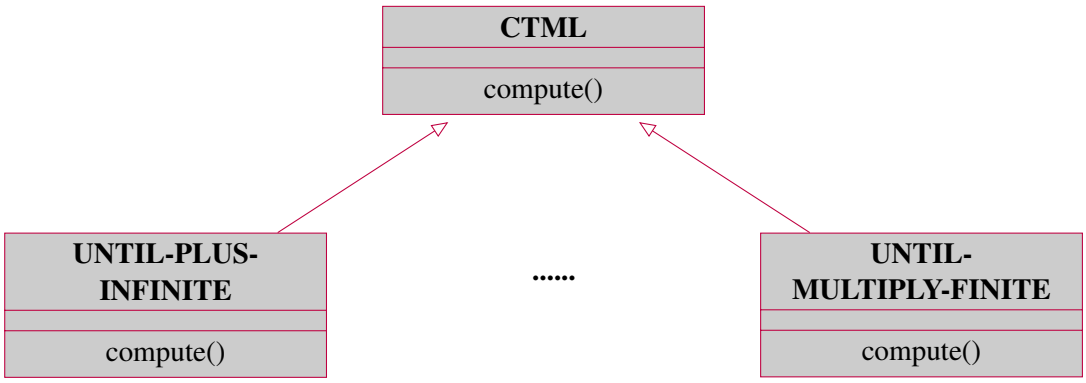


Figure 9.3: Polymorphic Design of CTML Software

The second design component is a scanner that is responsible for reading and processing input model along with a set of reward functions. The scanner gets ready for various data, such as the number of states, the number of edges, the transition probabilities, the set of atomic functions, etc., that are needed for processing each query. Also, the scanner reads in measure specifications in such a way that keeps track of two stacks, one is the *operator stack*, and the other is the *operands stack*. It processes the string of the measure in such a way that when the program sees a left parenthesis, it does nothing; when it sees an operator, it pushes the operator onto the operator stack; when it sees an operand such as an atomic function, it pushes it onto operand stack; when it sees a right parenthesis, it pops off an operator from the operator stack, pops associated number of operands from the operands stack, compute the operation, and push back the result onto the operand stack; the process continues until the operator stack is empty and one final result is left on the operand stack. The method is adapted from the well known Dijkstra's Shunting yard algorithm [26] for processing well parenthesized infix expressions. Table **??** presents the specific actions for each cases of character(s) seen so far.

9.2 Software Implementation

For the implementation, there are mainly two challenges that need to be resolved. One is the issue of storage due to the large state space involved, typically millions of states and tens of millions of edges. And another is the programming method for solving algebraic linear systems. For the first issue,

Table 9.1: Processing Infix Notation

Character(s)	Action
(	do nothing
MX, MU, MV, $*$, $+$, $\bowtie$ (operator)	push onto the operator stack
atomic functions (operand)	push onto the operands stack
)	1. pop an operator from the operator stack 2. pop the associated number of operands from the operand stack 3. invoke the operation 4. push back the result onto the operand stack

we use a data structure for sparse matrix storage; a set of primitive arrays that can be accessed fast. The second issue is resolved by using classic methods called Jacobi and Gauss-Seidel method. The following describes these two solutions in more detail.

9.2.1 Sparse Matrix Storage

For this work, our main input is a graph with probabilities attached to arcs, which are often represented by a real valued matrix $\mathbf{R}^{m\times n}$. To store a real valued matrix $\mathbf{R}^{m\times n}$, a brute-force way is to use a two dimensional array of reals that requires $\mathcal{O}(mn)$ memory space and time to perform vector-matrix or matrix-vector multiplication as shown in Algorithms 9.1 and 9.2, respectively. Better than that, we observe that most of our input models are *sparse*, in the sense that many entries are $zero$ values, as shown in Figure 9.7 followed by its corresponding matrix $\mathbf{P}$. If we can come up with a data structure such that it only stores non-zero entries, then we can obtain $\mathcal{O}(|\mathcal{S}| + |\mathcal{E}|)$ for both space and time complexity in vector-matrix multiplication, where $\mathcal{E}$ is the set of nonzero edges in a Markov process.

The basic idea for storing a sparse matrix [62] is that we create three arrays, say $ROWS$ with size equal to the number of states from the input model plus one (i.e., $|\mathcal{S}| + 1$). The $0th$ entry of $ROWS$ is

Algorithm 9.1 Compute Vector-Matrix Multiplication: $\mathbf{h} = \mathbf{xR}$.

$\mathbf{h} = \mathbf{0}$;
$\forall i, j$ such that $\mathbf{R}[i, j] \neq 0$ do
    $\mathbf{h}[j] = \mathbf{h}[j] + \mathbf{x}[i] \cdot \mathbf{R}[i, j]$;
End $\forall$

Algorithm 9.2 Compute Matrix-Vector Multiplication: $\mathbf{h} = \mathbf{Rx}$.

$\mathbf{h} = \mathbf{0}$;
$\forall i, j$ such that $\mathbf{R}[i, j] \neq 0$ do
    $\mathbf{h}[i] = \mathbf{h}[i] + \mathbf{R}[i, j] \cdot \mathbf{x}[j]$;
End $\forall$

initialized to 0; the content of $ROWS[1]$ is calculated by the number of outgoing edges from node 0 plus $ROWS[0]$; the content of $ROWS[2]$ is calculated by the number of outgoing edges from node 1 plus $ROWS[1]$, so on and so forth; the last entry of $ROWS[|\mathcal{S}|]$ stores the total number of edges. In practice, this last entry space is just the total number of edges if we know this number ahead of time. Also, for simplicity and efficiency purpose, we assume the input models are sorted by ROWS states starting from 0 (the idea would be the same if we assume the input model is sorted by $COLUMN$ states), which corresponds to the indices of $ROWS$ except for the last entry which we created to hold the value of the last row.

Now once we have the $ROWS$ array with proper contents, we create a $COLUMNS$ array with size equal to the number of edges. Obviously, for each node $i \in \mathcal{S}$, from $COLUMNS[ROWS[i]]$ until $COLUMNS[ROWS[i + 1]]$, it stores the index of state j in increasing order if there is an edge from state i to state j.

Finally, we create a $VALUES$ array with size equal to the number of edges; the array stores the actual real value (typically probabilities) corresponding to $COLUMNS[j]$ from $ROWS[i]$. Figures 9.4, 9.5, and 9.6 show examples for the contents of $ROWS$, $COLUMNS$, and $VALUES$ array, respectively, based on Figure 9.7. For in-depth handing of sparse matrix storage, interested readers should refer to [39, 50, 56, 60].

$ROWS$	0	1	2	3	4
$Contents$	0	2	5	8	10

Figure 9.4: Example of ROWS Array Contents Based on Figure 9.7.

$COLUMNS$	0	1	2	3	4	5	6	7	8	9
$Contents$	0	1	0	1	2	1	2	3	2	3

Figure 9.5: Example of COLUMNS Array Contents Based on Figure 9.7.

$$\mathbf{P} = \left[\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1/2 & 1/2 & 0 & 0 \\ b & 3/4 & 1/8 & 1/8 & 0 \\ c & 0 & 1/3 & 1/3 & 1/3 \\ d & 0 & 0 & 1/5 & 4/5 \end{array}\right]$$

Given the sparse matrix storage representation as described above, for a real valued matrix, we can now compute vector-matrix product or matrix-vector product, as shown in Algorithms 9.3 and 9.4, respectively, by utilizing the three arrays, namely $ROWS$, $COLUMNS$, and $VALUES$. Clearly, by taking advantage of sparse matrix storage, Algorithms 9.3 and 9.4 costs $O(|\mathcal{S}| + |\mathcal{E}|)$ spaces and operations in computing matrix-vector (or vector-matrix) multiplication.

Algorithm 9.3 Compute Vector-Matrix Multiplication Using Sparse Matrix Storage.

$r = 0$; // r is row index
$\mathbf{h} = \mathbf{0}$; // vector holding results
$\forall r < |ROWS|$ do
  $\forall j >= ROWS[i]$ and $j < ROWS[i+1]$
    $c = COLUMNS[j]$; // c is column index
    $\mathbf{h}[r] = \mathbf{h}[r] + \mathbf{x}[c] \cdot VALUES[j]$;
  End $\forall j$
End $\forall r$

VALUES	0	1	2	3	4	5	6	7	8	9
Contents	1/2	1/2	3/4	1/8	1/8	1/3	1/3	1/3	1/5	4/5

Figure 9.6: Example of VALUES Array Contents Based on Figure 9.7.

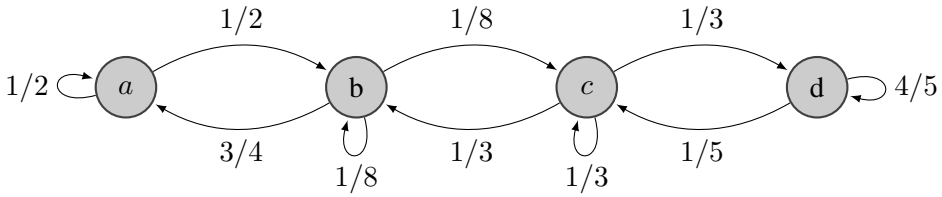


Figure 9.7: An Input Model Example

9.2.2 Solving Linear Systems

For CTML formulas with bounded time t, according to the algorithms presented in chapter 6, our implementation simply loops through t times over the modified version of Algorithms 9.3 and 9.4 such that it incorporates other necessary parameters (e.g. a real-valued diagonal matrix $\mathbf{F}$) and operations.

For CTML formulas with unbounded time t, however, the simple iteration method used for bounded time can no longer work efficiently as desired. Instead, we chose to use classic iterative methods known as *Jacobi* and *Gauss-Seidel*, for solving large linear system of the form $\mathbf{Rx} = \mathbf{h}$, or $\mathbf{xR} = \mathbf{R^T x} = \mathbf{h}$ [50]. The basic idea is that we start with a guessed solution $\mathbf{x}_0$, then we compute a sequence $\mathbf{x}_1, \cdots, \mathbf{x}_i, \mathbf{x}_{i+1}, \cdots, \mathbf{x}_n$ of the following form:

$$\mathbf{x}_{i+1} = \mathbf{R}\mathbf{x}_i + \mathbf{k} \quad (\text{where } \mathbf{R}, \mathbf{k} \text{ are known}),$$

Algorithm 9.4 Compute Matrix-Vector Multiplication Using Sparse Matrix Storage.

```
r = 0;                          // r is row index
h = 0;                          //vector holding results
∀r < |ROWS| do
    ∀j >= ROWS[i] and j < ROWS[i + 1]
        c = COLUMNS[j];         // c is column index
        h[c] = h[c] + x[r] · VALUES[j];
    End ∀j
End ∀r
```

until we reach a solution vector $\mathbf{x}_n$ (approximation of $\mathbf{x}$) that satisfies one of the following stopping criteria:

absolute precision:

$$\| \mathbf{x}_n[i] - \mathbf{x}_{n-1}[i] \| < \epsilon, \quad \text{for all } i \in \mathcal{S}.$$

or *relative precision*:

$$\left| \frac{\mathbf{x}_n[i] - \mathbf{x}_{n-1}[i]}{\mathbf{x}_n[i]} \right| < \epsilon, \quad \text{for all } i \in \mathcal{S}.$$

where ϵ is a threshold value for determining whether we should stop iterations.

Take Figure 9.7 as an example, suppose we want to solve the following linear equation.

$$\boldsymbol{\pi}(\mathbf{P} - \mathbf{I}) = \mathbf{0}, \quad \text{or} \quad (\mathbf{P} - \mathbf{I})^T \boldsymbol{\pi} = \mathbf{0}$$

Then, by plug in the probability transition matrix $\mathbf{P}$, we have

$$(\mathbf{P} - \mathbf{I})^T = \begin{bmatrix} -5/6 & 3/4 & 0 & 0 \\ 5/6 & -7/8 & 1/3 & 0 \\ 0 & 1/8 & -2/3 & 1/5 \\ 0 & 0 & 1/3 & -1/5 \end{bmatrix} \quad \boldsymbol{\pi} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

Multiplying out, we obtain the following system of equations:

$$a = (3/4b - 0)/(5/6) = 8/5b \tag{9.1}$$

$$b = (5/6a + 1/3c - 0)/(7/8) = 20/21a + 8/21b \tag{9.2}$$

$$c = (1/8b + 1/5d - 0)/(2/3) = 3/16b + 3/10d \tag{9.3}$$

$$d = (1/3c - 0)/(1/5) = 5/3c \tag{9.4}$$

Set $\boldsymbol{\pi}_0 = [1/4,\ 1/4,\ 1/4,\ 1/4]$, we have

Iteration 1:

$$a_1 = 2/5 \tag{9.5}$$

$$b_1 = 5/21 + 5/21 = 10/21 \tag{9.6}$$

$$c_1 = 3/64 + 3/40 = 39/320 \tag{9.7}$$

$$d_1 = 5/3c = 5/12 \tag{9.8}$$

Note that the sum of $(a_1 + b_1 + c_1 + d_1)$ is not 1, so we need to do *normalization* such that each value of a_1, b_1, c_1, d_1 is divided by the sum value of $(a_1 + b_1 + c_1 + d_1)$. By doing so, in this case we get

$$a_1 \approx 0.28274,\ b_1 \approx 0.33659,\ c_1 \approx 0.086147,\ d_1 \approx 0.29452.$$

Then, we keep computing iteration 2, iteration 3, $\cdots$. Now if we choose to use *Jacobi* iteration method, then we need to keep two vectors for $\boldsymbol{\pi}_{i+1}$ and $\boldsymbol{\pi}_i$, respectively, since $\boldsymbol{\pi}_{i+1}$ is computed only after all values of $\boldsymbol{\pi}_i$ becomes available. For example, by using Jacobi iteration method, following iteration 1, we have

Iteration 2:

$$a_2 = (3/4 * b_1)/(5/6) \tag{9.9}$$

$$b_2 = (5/6 * a_1 + 1/3 * c_1)/(7/8) \tag{9.10}$$

$$c_2 = (1/8 * b_1 + 1/5 * d_1)/(2/3) \tag{9.11}$$

$$d_2 = (1/3 * c_1)/(1/5) \tag{9.12}$$

In contrast, if we choose to use *Gauss-Seidel* iteration method, then we can use values that are newly computed immediately. For example, by using Gauss-Seidel method, following iteration 1, we have:

Iteration 2:

$$a_2 = (3/4 * b_1)/(5/6) \tag{9.13}$$

$$b_2 = (5/6 * a_2 + 1/3 * c_1)/(7/8) \tag{9.14}$$

$$c_2 = (1/8 * b_2 + 1/5 * d_1)/(2/3) \tag{9.15}$$

$$d_2 = (1/3 * c_2)/(1/5) \tag{9.16}$$

Either using Jacobi or Gauss-Seidel, we keep iterating until the vector converges. With this example, our solution should be:

$$\boldsymbol{\pi} \approx \boldsymbol{\pi}_n = [3/7,\ \ 2/7,\ \ 3/28,\ \ 5/28].$$

For more details about solving linear systems, interested readers should refer to [50, 60]. In our CTML tool, both methods of Jacobi and Gauss-Seidel are used in implementing the algorithms that we discussed in previous Chapter for CTML formulas.

9.3 asCTML Software Translator

The asCTML translator is a separate tool from CTML. The translator reads an MAMC input file along with a set of the atomic state+action formulas, and output a DTMC plus a corresponding set of atomic state formulas for CTML. Figure 9.8 gives an example format for the input file for the MAMC structure shown in Figure 8.2(a). The output format is the same as the example shown in Figure 9.2. The implementation is tedious but straightforward. It merely follows the translation algorithm 8.2 for setting the states and edges and probabilities and action names.

The basic idea is that it first creates a DTMC state array, with its size equal to the number of arcs from MAMC. Then it scans through the model and stores the accumulated number of the DTMC edges in each slot of the DTMC state array. Meanwhile, it stores the corresponding value (for later translation) in MAMC state array, MAMC arcs array, MAMC probability array, and MAMC action array, in a way similar to those described in sparse matrix storage for rows 9.4, columns 9.5, and probabilities 9.6. At the end of scanning, we get the total number of arcs for the translated DTMC. Given the size of the DTMC arcs, we create a DTMC arcs array, a DTMC probability array, and a DTMC action array. The

next step is to process the four MAMC arrays discussed above and store the translated information into the corresponding DTMC arrays. At the last step, we go through these DTMC arrays, and print the DTMC along with the atomic formulas.

9.4 Overall Discussion and Software Test

Our prototype tools are implemented in Java, for the evaluation of CTML and asCTML languages. The testing environment we used is a desktop computer with a 2.5 Ghz Intel core 2 duo processor and 4GB of 667 MHz RAM, running MacOS X, and the Java virtual machine for JDK version 1.6. The testing strategy is that we first plug in small well known Markov chain models such as "Luck of Fortune" into our tool together with various CTML and asCTML queries, then we verify the outputs of the software against the numerical solutions computed by hand. Because of the sensitivity of the floating point precisions of numerical values, after running several dozens of such small cases for each different formulas and getting all consistent answers, we believe that our implementation is correct and robust. Then, we move on to large models for automatic blind testing.

```
MAMC
STATES 4
INIT
  0 : 1.0
ARCS   9
  0 : 2 : 0.5 : a
  0 : 1 : 0.2 : c
  0 : 1 : 0.3 : b

  1 : 2 : 0.5 : b
  1 : 2 : 0.5 : c

  2 : 3 : 1.0 : a

  3 : 1 : 0.2 : a
  3 : 1 : 0.2 : d
  3 : 3 : 0.6 : c

  2 : 2 : 1.0
  3 : 3 : 1.0
END

f
  0 : a : 1.0
  0 : b : 3.0
  1 : c : 2.0
  2 : a : 1.0
  3 : c : 2.0
end_f

 l
  0 : c : 1.0
  1 : b : 1.0
  3 : a : 1.0
end_l

g
 3 : d : 1.0
end_g


MEASURE
    (f until_plus g)
```

Figure 9.8: An Example Input Model Format for asCTML Translator

CHAPTER 10. APPLICATION EXAMPLE

In this chapter, we demonstrate the use of CTML and asCTML through the analysis of two application examples. In case users are interested in dependability-performance related queries that do not depend on actions, then CTML applies. Otherwise, if the associated queries are action dependent, then asCTML applies; in this case, the input model and the asCTML specifications can be reduced to DTMC and CTML, as we have shown in chapter 8. Since many existing models have named actions, we believe asCTML has wide applications. Compared with CTML, asCTML can quantify much wider sets of paths characterized by a variety of actions.

10.1 University Graduation Example

In this section, we deploy a small model structure for the discussion of example asCTML and CTML queries. With the example of small model structure, we tend to show users more clearly the quantification of action driven paths; how action dependent queries can be expressed by asCTML but not CTML; and how action independent queries can be expressed by CTML, and their syntactic similarities. More importantly, through the small application example, we tend to deliver the overall specification idea of asCTML with respect to CTML.

The *University Graduation* example as shown in Figure 10.1 is from [38]. Each year, at a fictitious four year undergraduate university, a student has probability p to move up to a higher grade, has probability r to repeat for the same grade, and probability q to drop the school. The following presents interesting queries for this model.

1. *"What's the average number of years of repeating at the freshman level?"* First of all, the query requires to count the `repeat` action at the freshman year only. So we define an atomic formula $repeats\text{-}fr$ such that it has value 1 on $(fr, repeat)$, and 0 on all others. Also, for this model, there

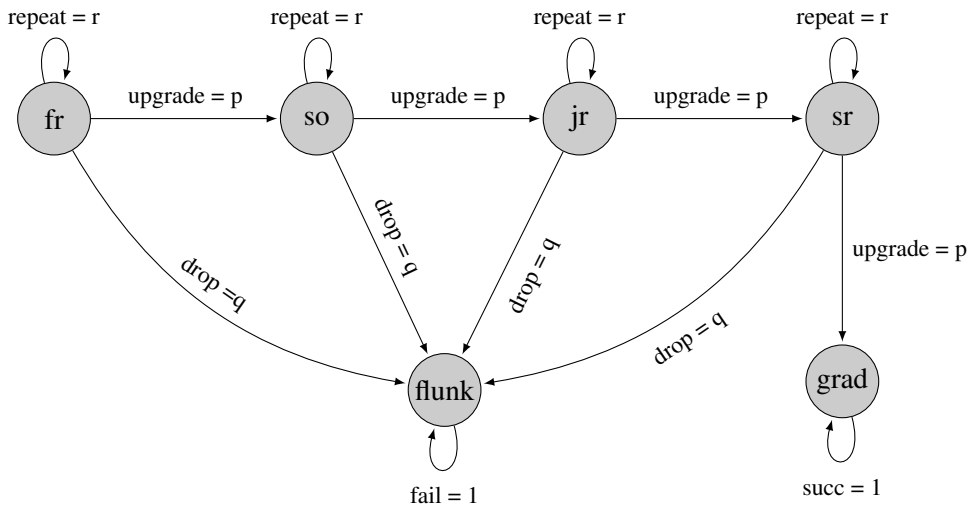


Figure 10.1: University Graduation Example

are only two destinations which are either finish successfully indicated by `grad` or `flunk`. So we define another atomic formula *finish* such that it has value 1 on $(grad, succ)$ and $(flunk, fail)$, and 0 otherwise. The query can then be expressed as

$$M\ repeats\text{-}fr\ U_+\ finish$$

In general, this type of query is not expressible by CTML. $repeats\text{-}fr(fr, repeat)$ to some other positive value other than 1. In this particular case, however, we can use the CTML expression $M\ fr\ U_+\ (1 - fr)$ to get the average number of years spent as a freshman, and then we subtract one to get the number of repeats.

2. *"What's the average number of repeating at all levels of grades given that students graduate successfully?"* Unlike the previous query, this query asks for the quantification of the repeating behavior at all levels of grades, so we define an atomic formula $repeats$ such that $repeats(\{fr, so, jr, sr\}, repeat) = 1$ and 0 for all others. Also, we need an atomic formula $graduate(grad, succ) = 1$ and 0 otherwise. Then the query can be expressed as

$$M repeats\ U_+\ graduate$$

By the definition of conditional expectation equation 2.5, the result of the expression must be divided by the quantity of $M\,one\,U_{\times}\;graduate$. The query is not expressible by CTML in general.

3. *"What's the probability to quit the school after sophomore year?"* Let $quit\text{-}after\text{-}soph$ be an atomic formula such that $quit\text{-}after\text{-}soph(\{so, jr, sr\},\, drop) = 1$, and 0 for all others. Then the query can be expressed as

$$M\,one\,U_{\times}\;quit\text{-}after\text{-}soph$$

This query is expressible by CTML as

$$M\,one\,U_{\times}(M\;so\,U_{\times}\;\mathit{flunk})$$

and by PLTL as

$$P_{?=}F(so\,U\;\mathit{flunk})$$

In this case, asCTML requires one path operator $U_{\times}$, whereas both CTML and PLTL requires two path operators.

4. *"What's the probability of graduation in 6 years?"* Let $graduate$ be an atomic formula with $graduate(grad, succ) = 1$ and 0 otherwise. Then the query can be expressed as

$$M\;one\,U_{\times}^{\leq 6}\;graduate$$

This query is expressible by CTML, with the same syntax as for asCTML, except that with CTML the atomic formulas $graduate$ and one quantify the corresponding states only, rather than the (state, action) pairs.

5. *"Given that a student reached the junior state and never repeated a year before, what's the probability that the student will graduate in 3 years?"* This query requires to only quantify the `upgrade` action at the freshman and sophomore level as the conditional statement imposes. Let $no\text{-}repeat\text{-}soph(\{fr, so\}, upgrade) = 1$, and 0 otherwise. Let $graduate(grad, succ) = 1$ and

0 otherwise. Let $junior(jr, \{repeat, upgrade, drop\}) = 1$, and 0 otherwise. Then this query can be expressed as

$$M\ no\text{-}repeat\text{-}soph\, U_{\times}(junior \cdot M\ one\, U_{\times}^{\leq 3} graduate)$$

and the result must be divided by $M\ no\text{-}repeat\text{-}soph\, U_{\times}\ junior$, according to the conditional probability definition 2.2. Since the query depends on actions, it is not expressible by CTML.

6. *"What's the probability that students drops the school before junior year?"* This question is equivalent to, the probability that a student never becomes a junior. Let $junior$ be an atomic formula such that $junior(jr, \{repeat, upgrade, drop\}) = 1$, and 0 otherwise. Then this query can be expressed as:

$$1-(M\ one\, U_{\times}\ junior)$$

This query is expressible by CTML, since it does not depend on actions.

We note that the main contribution of asCTML is that it can quantify all kinds of combinations of actions along the paths. In case there are multiple actions between the same pair of states, the idea of the specification would be the same.

We now discuss experimental results for the queries. First of all, we come up with an MAMC model structure for the University Graduation example, and a set of atomic state+ action formulas for the queries. we set the probabilities p, r, q to be $0.1, 0.8$, and 0.1, respectively. The input format and some atomic formulas are shown in Figure 10.2. The MAMC model and the atomic state+action formulas are then translated into DTMC model and CTML atomic state formulas, which are then plugged into CTML software for evaluation of each queries. The asCTML specification for each query can either be created at run time or written at the end of the model. Also, since the specifications are syntactically the same for asCTML and CTML. they can either be included in MAMC model and carried over to the DTMC model or be put in the translated DTMC model directly. Table 10.1 presents the numerical results each query discussed above.

```
 MAMC
 STATES 6
 INIT
 0 : 1
 ARCS 14
   0 : 0 : 0.1 : repeat
   0 : 1 : 0.8 : upgrade
   0 : 5 : 0.1 : drop

   1 : 1 : 0.1 : repeat
   1 : 2 : 0.8 : upgrade
   1 : 5 : 0.1 : drop

   2 : 2 : 0.1 : repeat
   2 : 3 : 0.8 : upgrade
   2 : 5 : 0.1 : drop

   3 : 3 : 0.1 : repeat
   3 : 4 : 0.8 : upgrade
   3 : 5 : 0.1 : drop

   4 : 4 : 1.0 : succ

   5 : 5 : 1.0 : fail
END

repeatsfr
  0 : repeat : 1.0
end_repeatsfr

finish
  4 : succ : 1.0
  5 : fail : 1.0
end_finish

repeats
  0 : repeat : 1.0
  1 : repeat : 1.0
  2 : repeat : 1.0
  3 : repeat : 1.0
end_repeats

quitaftersoph
  1 : drop : 1.0
  2 : drop : 1.0
  3 : drop : 1.0
end_quitaftersoph
```

Figure 10.2: An MAMC Example for The University of Graduation and Some Atomic State+Action Formulas

Table 10.1: Numerical Results for the asCTML Queries on the University Graduation Example (with probabilities $r = 0.1$, $p = 0.8$, and $q = 0.1$)

#	Query	Initial State	Numerical Result
1	$M\ repeats\text{-}fr\ U_{+}\ finish$	fr	0.11111
2	$\frac{M\ repeats\ U_{+}\ graduate}{M\ one\ U_{\times}\ graduate}$	fr	0.44443
3	$M\ one\ U_{\times}\ quit\text{-}after\text{-}soph$	fr	0.26459
4	$M\ one\ U_{\times}^{\leq 6}\ graduate$	fr	0.57344
5	$\frac{M\ no\text{-}repeat\text{-}soph\ U_{\times}\ (junior \cdot M\ one\ U_{\times}^{\leq 3}\ graduate)}{M\ no\text{-}repeat\text{-}soph\ U_{\times}\ junior}$	fr	0.64000
6	$1 - M\ one\ U_{\times}\ junior$	fr	0.20988

10.2 Dining Philosopher Example

The classic dining philosophers model [27] as shown in Figure 10.3 is similar to the one from [51]. The model is described in stochastic Petri net formalism [1, 15]. For the sake of space, the picture only shows a single philosopher. In experimentation, we used around 3-10 dining philosophers (corresponding to 3-10 copies of the subnets of the connected philosophers). The figure shows the initial state of the dining philosopher, where a token in place idl suggests that the philosopher is in thinking state; a token in place $fork_i$ suggests $fork_i$ is available. Now imagine that we have multiple philosophers connected, sitting in a round table and sharing the left fork with a philosopher sitting on the left hand side, and sharing the right fork with a philosopher sitting on the right hand side. After a while, a philosopher may become hungry. In order to eat, the philosopher must first obtain a left and right fork, which means he/she must "fire" transitions $getl$ and $getr$. When the philosopher gets one fork, he/she must wait for the other fork before he/she can eat. After eating, the philosopher releases forks via transition $release$. This model is well known for having exactly two deadlocked states, corresponding to every philosopher holding their left fork, and to every philosopher holding their right fork.

In the following, we describe performance measures through example queries followed by the corresponding CTML and asCTML specifications. Due to the large size, it is neither realistic nor necessary for us to show whether an asCTML query is expressible by CTML. Rather, we choose one that can express the query more intuitively.

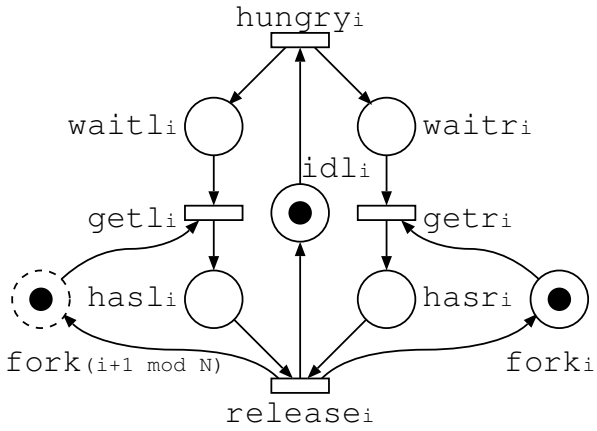


Figure 10.3: SPN Model of A Single Dining Philosopher.

1. *"How much food will philosopher 1 consume, on average, before reaching deadlock?"* We define an atomic formula *food* that specifies, for each state, how much food philosopher 1 consumes per unit of time (with value 0 when philosopher 1 is not eating); and we define an atomic function *deadlock* with value 1 for the deadlocked states, and 0 otherwise. The query can then be written in CTML as

$$M\ food\ U_+\ deadlock.$$

2. *"How much food will philosopher 1 consume in the first t time units, on average?"* Consistent with the above query, we can write this in CTML as

$$M\ food\ V_+^{\leq t}\ deadlock,$$

since no food is consumed in the deadlock states. Alternatively, we can write

$$M\ food\ V_+^{\leq t}\ zero.$$

3. *What is the probability that philosopher 1 will obtain the left fork within t time units?* This is similar to the previous query, except we use a time–bounded formula:

$$M \ one \, U_{\times}^{\leq t} \ left,$$

where *left* returns 1 only for states where philosopher 1 has the left fork.

4. *What is the probability that phil 1 eventually eats?* First, note that for any state formula g, the formula

$$M \ one \, U_{\times} \ g$$

computes the sum over all states s with $g(s) > 0$, the probability that s is reached before any other state s' with $g(s') > 0$, multiplied by $g(s)$. We can therefore express the above query as

$$M \ one \, U_{\times} \ eat$$

where atomic function *eat* returns 1 for any state where philosopher 1 is eating (i.e., has both forks), and is 0 otherwise.

5. *"Given that philosopher 1 obtains the left fork, what is the probability that philosopher 1 eventually eats?"* To determine this quantity, we use the fact that $\Pr\{B \mid A\} = \Pr\{B \cap A\} / \Pr\{A\}$. Thus we first determine the probability that philosopher 1 obtains the left fork, and then eats.

   Since before philosopher 1 eats, he/she must obtain the left fork, we can write this in CTML as

$$M \ one \, U_{\times} \ (M \ left \, U_{\times} \ eat). \tag{10.1}$$

   To allow philosopher 1 to release the left fork before eating, we would write

$$M \ one \, U_{\times} \ ((M \ one \, U_{\times} \ eat) \cdot left) \tag{10.2}$$

   instead. Note that the former is preferred when possible, since the linear system for $M \ left \, U_{\times} \ eat$ has smaller dimension than the one for $M \ one \, U_{\times} \ eat$. These types of queries (in general) are

not expressible with PCTL, even allowing the top–most $P_{=?}$ operation as described in PRISM [44]. However, they are expressible in PLTL as $P_{=?}[F(\mathit{left}\ U\, \mathit{eat})]$ and $P_{=?}[F(\mathit{left} \wedge F\, \mathit{eat})]$, respectively. Alternatively, we can note that, for this model, before philosopher 1 eats, he/she must obtain the left fork, and write this as

$$M\ \mathit{one}\, U_{\times}\ \mathit{eat}.$$

Regardless of the formula used, we must divide the result by the quantity of $M\ \mathit{one}\ U_{\times}\ \mathit{left}$, which is the probability that philosopher 1 obtains the left fork, for some starting state s.

6. *"If philosopher 1 just picked up left fork, then what is the expected time until deadlock?"* First, we determine the expected time until deadlock starting from each possible state. Then, we filter out all but the states where philosopher 1 has the left fork. We then sum over all left fork states, the probability to reach that one "first" multiplied by the expected time to deadlock starting from that state. This gives us the CTML expression

$$M\ \mathit{one}\, U_{\times}(\mathit{left} \cdot (M\ \mathit{one}\, U_{+}\ \mathit{deadlock})). \tag{10.3}$$

By the definition of conditional expectation equation 2.5, we must divide the result by $M\ \mathit{one}\, U_{\times}\ \mathit{left}$.

7. *"What is the probability that philosopher 1 never eat before deadlock?"* Since we do not count actions that indicate philosopher 1 is $eating$, we define $never\text{-}eat$ to be an atomic formula such that it has value 0 on all pairs of $(s, release_1)$ where s can be any state at which the transition $\texttt{release}_1$ is enabled, and 1 otherwise. Let $deadlock$ be defined as above. The query can then be expressed in asCTML as

$$M\ never\text{-}eat\, U_{\times}\ deadlock$$

8. *"How much food will philosopher 1 consume, on average, between obtaining and releasing the forks?"* We define an atomic formula *food* as above. We also define an atomic formula $release$ such that it has value 1 on all pairs $(s, release_1)$ and 0 otherwise. The query can then be

expressed in asCTML as

$$M\ food\, U_+\ release$$

9. *"What is the probability that philosopher 1 triggered the deadlock (i.e., was the last philosopher to pick up a fork)?"* Let $deadlock\text{-}trigger$ be an atomic formula that has value 1 on all pairs of $(s, getl_1)$ and $(s, getr_1)$. Let $deadlock$ be as previously defined. The query can then be expressed in asCTML as

$$M\ one\, U_\times(deadlock\text{-}trigger \cdot MX\, deadlock)$$

If an action of $\texttt{getl}_1$ or $\texttt{getr}_1$ triggers the `deadlock`, then `deadlock` must be the next state. As such, $\phi = (deadlock\text{-}trigger \cdot MX\, deadlock)$ must have positive values, then $M\ one\, U_\times \phi$ gives the probability that philosopher 1 triggered the deadlock.

10. *"How many times will philosopher 1 eat, on average, before reaching deadlock?"* Let $count\text{-}eat$ be an atomic formula such that it has value 1 on all pairs of $(s, release_1)$ where s can be any state at which `release_1` is enabled, and 0 otherwise. Let $deadlock$ be defined as before. This query can be expressed in asCTML as

$$M\, count\text{-}eat\, U_+\ deadlock$$

For the dining philosopher model, we are mainly interested in exploring the various model size for different number of philosophers and running time of the example queries. To test large models, we deploy SMART [16] as our front-end engine. Specifically, we plug in the dining philosopher of stochastic Petri-net benchmark models into the SMART tool, and generate CTMCs, then we study their embedded DTMC. For asCTML queries, the MAMC models are obtained by having a set of named transitions associated with the DTMCs. That is, each edge has a named action. The MAMC models are then fed into the asCTML translator for the conversion to DTMC and CTML. Table **??** lists the translated DTMC sizes for each of the corresponding MAMC model along with the CPU time for the typical translation time of the model. The converted DTMC and CTML queries can then be plugged into the CTML software for evaluation. Note that in case the queries are all CTML formulas (that is,

action independent), then we can skip the step of (asCTML to CTML) translation. The tool has been put to experiment on various number of dining philosophers until the computer is run out of memory when the model is too large. For the case of 10 dining philosophers, it produces a DTMC with 1,860,498 states for CTML queries. For the nested formula of type $MU_{\times}(MU_{+})$, we first compute its inner formula, and we have $|\mathcal{S}_g|$ = 2 for the two deadlocked states and $|\mathcal{S}_n|$ = 0; the query requires solution of two linear systems of dimension $|\mathcal{S}_z|$= 1,860,496. Using Gauss Seidel and a relative precision of 10^6, this requires roughly 689 seconds of CPU time. Then, we compute the outer formula, $MU_{\times}$. For a typical example in which we say a dining philosopher has obtained the left fork, with $|S_n| = 1$ from which the dining philosopher will never obtain the right fork, leaving $|\mathcal{S}_z| = 1,346,268$. This linear system is solved in about 32 seconds. So the typical nested formula $MU_{\times}(MU_{+})$ runs about 721 seconds in total. Table **??** presents the numerical results and CPU time for both the CTML and asCTML queries we discussed.

Table 10.2: Results of Translated Size of DTMCs ($-$: out of memory).

Model	**DTMC for CTML**		**Translated DTMC for asCTML**		**CPU Time** (seconds)
	# states	# edges	# states	# edges	
5 phils	1,364	6,377	6,377	29,372	0.92
6 phils	5,778	32,408	32,408	179,552	1.35
7 phils	24,476	160,155	160,155	1,037,010	3.33
8 phils	103,682	775,338	775,338	5,745,082	17.65
9 phils	439,204	3,694,925	3,694,925	30,832,400	185.34
10 phils	1,860,498	17,391,050	17,391,052	161,376,412	–

Table 10.3: Numerical Results for Selected CTML Queries of Different DTMC sizes (−: out of memory).

#	Query	Number of Philosophers				
		6 phils	**7 phils**	**8 phils**	**9 phils**	**10 phils**
1	$M\,food\,U_{+}\;deadlock$	14.5476	20.0333	26.3501	33.4931	41.4595
	CPU (CTML)	1.1 sec	3.6 sec	17.1 sec	89.5 sec	564.4 sec
2	$M\,food\,V_{+}^{\leq 300}\;deadlock$	13.5434	17.0743	20.0597	22.4465	24.3176
	CPU (CTML)	0.9 sec	1.3 sec	2.7 sec	7.3 sec	29.5 sec
3	$M\,one\,U_{\times}^{\leq 20}\,left$	0.6733	0.5985	0.5385	0.4818	0.4341
	CPU (CTML)	0.7 sec	1.1 sec	1.8 sec	4.2 sec	15.8 sec
	$M\,one\,U_{\times}\,left$	0.9501	0.9620	0.9703	0.9764	0.9809
	CPU (CTML)	0.8 sec	1.2 sec	2.3 sec	7.0 sec	33.1 sec
4	$M\,one\,U_{\times}\,eat$	0.9002	0.9239	0.9406	0.9527	0.9617
	CPU (CTML)	0.7 sec	1.3 sec	2.7 sec	7.9 sec	39.1 sec
5	$M\,one\,U_{\times}(M\,left\,U_{\times}\;eat)$	0.9002	0.9239	0.9406	0.9527	0.9617
	CPU (CTML)	0.9 sec	1.5 sec	3.4 sec	11.4 sec	58.5 sec
	$\frac{M\,one\,U_{\times}(M\,left\,U_{\times}\;eat)}{M\,one\,U_{\times}\,left}$	0.9475	0.9605	0.9694	0.9758	0.9805
6	$M\,one\,U_{\times}(left\cdot M\,one\,U_{+}\;deadlock)$	112.6291	160.2409	217.0598	283.3792	359.4763
	CPU (CTML)	1.4 sec	3.8 sec	17.8 sec	91.9 sec	585.6 sec
	$\frac{M\,one\,U_{\times}(left\cdot M\,one\,U_{+}\;deadlock)}{M\,one\,U_{\times}\,left}$	118.5445	166.5706	223.7038	290.2285	366.4760
7	$M\,never\text{-}eat\,U_{\times}\,deadlock$	0.0998	0.0761	0.0579	0.0473	-
	CPU (asCTML)	1.4 sec	2.8 sec	11.1 sec	66.1 sec	-
8	$M\,food\,U_{+}\,release$	2.1416	2.7209	3.3036	3.8877	-
	CPU (asCTML)	1.6 sec	3.8 sec	17.3 sec	102.3 sec	-
9	$M\,one\,U_{\times}(deadlock\text{-}trigger\cdot MX\,deadlock)$	0.1667	0.1429	0.1250	0.1111	-
	CPU (asCTML)	2.8 sec	11.6 sec	86.3 sec	569.9 sec	-
10	$M\,count\text{-}eat\,U_{+}\,deadlock$	4.8853	5.9282	6.9990	7.2867	
	CPU (asCTML)	3.8 sec	19.5 sec	180.6 sec	1131.2 sec	-

CHAPTER 11. CONCLUSION AND FUTURE RESEARCH

To summarize, up to this point, model checking techniques have gone through three generations. The first generation is represented by the classic temporal logics such as CTL and LTL and the underlying model is typically a Kripke structure. The second generation is probabilistic model checking; they are represented by PCTL, PLTL, and CSL, and the underlying model is typically a Markov chain structure, either discrete time or continuous time. From the first generation to the second generation, model checking techniques has transformed from qualitative analysis to quantitative analysis in limited way. The limited quantitative analysis of probabilistic model checking come from the following facts: first, the quantity is a probabilistic value between 0 and 1; second, probabilistic logics such as PCTL and CSL cannot nest real values, since they are by nature logics and produce true/false values only; though PLTL can output real values between 0 and 1, its overall computational complexity is exponential in the size of the formula and polynomial in the size of the model. The third generation, represented by CTML, or asCTML for a later extension, is relatively a more seamless formal language towards the unification of model checking and performance evaluation; it is classified as third generation for the fact that, the language adopts model checking's specification style and retains the original model checking's expressive power, yet it is no longer a logic, because it can handle and nest any real valued quantities between 0 and infinity rather than between 0 and 1. The language is currently supported by a set of algorithms and software tools. The following discusses potential future work.

Compare to CTML, asCTML can answer questions such as: *What is the expected number of times that a particular event occurs before deadline t?* that can not be answered by CTML, because CTML can not handle weighted arcs along paths, nor does it support multiple actions. While asCTML is powerful, it can not handle queries like *What is the probability that a given path formula has accumulated value larger than x?* This is another type of useful query related to performance measures and was

recently addressed in [2], which presents an operator for a path formula to be able to check against the boundary real-valued interval J. One of the problems is that the algorithm is exponential in the size of the graph if the state-based rewards are real values. This makes the approach less practical due to large state space for a typical model structure.

Another direction for future research is to apply asCTML or a similar language for continuous time Markov chains (CTMCs) or semi-Markov processes (SMPs). For unbounded until and unbounded weak until queries, this is fairly straightforward: since CTML can handle real–valued state formulas, we can instead analyze the embedded DTMC using CTML, and scale the state formulas by the expected time spent in each state. The time bounded versions of these formulas are not so straightforward to handle, and will likely require significant changes. For CTMC, one of the main difficulties is how the reward values shall be accumulated, since the time distribution is no longer $Const(1)$ as in DTMC. If we calculate an embedded DTMC via uniformization, then the quantity varies with how q is chosen. SMP is essentially for general distributions that are not limited to exponential; as like CTMC, however, the analysis for bounded until is the main challenge. Helpful inspirational works can be found here [7, 35, 48], though they are all similar to CSL.

In addition to CTMCs and SMPs, another type of Markov processes that are of our interest is Markov Decision Processes (MDPs). MDPs are essentially extension of Markov chains with rewards; the main difference is the addition of a set of non-deterministic actions. If a *policy* is assumed to be given, then MDPs can be reduced to Markov chains and analyzed by model checking formalisms similar to PCTL. This idea is presented in [23, 46]. More specifically, [23] handles a probabilistic structure where the duration time is either 0 or 1 between state to state transitions. [46] extended the idea by allowing the duration time to be an arbitrary natural number between state to state transitions. While [24, 45] extended the previous approaches with real values between state to state transitions for computing a set of measures so called *probabilistic reachability* and *expected reachability* [43], they are not supported by a formal specification language, and therefore limited in handling complex measures such as nested formulas. On the other hand, asCTML computes a wide variety of complex formulas for DTMCs/MAMCs, hence working on MDPs can be a potential future work for a broader measures in the domain of optimization and concurrency.

In chapter 7, we have already shown that CTML covers the right recursive PLTL formulas that cannot be expressed by PCTL. In the future, we wish to further extend asCTML such that it covers pCTL* [3], which is a unification of PCTL and PLTL. In that case, however, the time complexity will become exponential in the length of the formula.

Bibliography

[1] Ajmone Marsan, M., Conte, G., and Balbo, G. (1984). A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122.

[2] Andova, S., Hermanns, H., and Katoen, J. P. (2003). Discrete-time rewards model-checked. In Larsen, K. G. and Niebert, P., editors, *Formal Modelling and Analysis of Timed Systems (FORMATS 2003)*, volume 2791 of *Lecture Notes in Computer Science*, pages 88–104. Springer Verlag.

[3] Aziz, A., Singhal, V., and Balarin, F. (1995). It usually works: The temporal logic of stochastic systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 155–165, London, UK. Springer-Verlag.

[4] Baarir, S., Beccuti, M., Cerotti, D., De Pierro, M., Donatelli, S., and Franceschinis, G. (2009). The GreatSPN tool: Recent enhancements. *SIGMETRICS Perform. Eval. Rev.*, 36(4):4–9.

[5] Baier, C., Cloth, L., Haverkort, B. R., Kuntz, M., and Siegle, M. (2007). Model checking Markov chains with actions and state labels. *IEEE Transactions on Software Engineering*, 33:209–224.

[6] Baier, C., Haverkort, B., Hermanns, H., and Katoen, J.-P. (2003). Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541.

[7] Baier, C., Haverkort, B. R., Hermanns, H., and Katoen, J.-P. (2000). On the logical characterisation of performability properties. In *Automata, Languages and Programming*, pages 780–792.

[8] Balbo, G. (2000). Introduction to stochastic Petri nets. In Brinksma, E., Hermanns, H., and Katoen, J.-P., editors, *European Educational Forum: School on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 84–155. Springer.

[9] Beaudry, D. M. (1978). Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, C-27(6):540–547.

[10] Bertsekas, D. P. and Tsitsiklis, J. N. (1991). An analysis of stochastic shortest path problems. *Math. Oper. Res.*, 16(3):580–595.

[11] Bianco, A. and de Alfaro, L. (1995). Model checking of probabalistic and nondeterministic systems. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 499–513, London, UK. Springer-Verlag.

[12] Borgerson, B. R. and Freitas, R. F. (1975). A reliability model for gracefully degrading and standby-sparing systems. *IEEE Trans. Comput.*, 24(5):517–525.

[13] Çinlar, E. (1975). *Introduction to stochastic processes*. Prentice-Hall, Englewood Cliffs, NJ.

[14] Chung, K. L. (1979). *Elementary Probability Theory with Stochastic Processes*. Undergraduate Texts in Mathematics. Springer-Verlag, Orlando, 3 edition.

[15] Ciardo, G. (1989). *Analysis of large stochastic Petri net models*. PhD dissertation, Duke University, Durham, NC.

[16] Ciardo, G., R. L. Jones, I., Miner, A. S., and Siminiceanu, R. I. (2006). Logic and stochastic modeling with SMART. *Perform. Eval.*, 63(6):578–608.

[17] Ciardo, G. and Trivedi, K. S. (1993). A decomposition approach for stochastic reward net models. *Perf. Eval*, 18:37–59.

[18] Clark, G., Gilmore, S., and Hillston, J. (1999). Specifying performance measures for PEPA. In *ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, pages 211–227, London, UK. Springer-Verlag.

[19] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263.

[20] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press.

[21] Cohn, D. L. (1980). *Measure Theory*. Birkhäuser, Boston.

[22] Courcoubetis, C. and Yannakakis, M. (1995). The complexity of probabilistic verification. *J. ACM*, 42(4):857–907.

[23] de Alfaro, L. (1997). Temporal logics for the specification of performance and reliability. In *STACS'97: Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, pages 165–176, London, UK. Springer-Verlag.

[24] de Alfaro, L. (1999). Computing minimum and maximum reachability times in probabilistic systems. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 66–81, London, UK. Springer-Verlag.

[25] de Alfaro, L., Faella, M., Henzinger, T. A., Majumdar, R., and Stoelinga, M. (2005). Model checking discounted temporal properties. *Theor. Comput. Sci.*, 345(1):139–170.

[26] Dijkstra, E. W. (1961). Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60. Technical Report MR 34/61.

[27] Dijkstra, E. W. (1971). Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138.

[28] Donatelli, S., Haddad, S., and Sproston, J. (2007). CSL^{TA}: an expressive logic for continuous-time Markov chains. *Quantitative Evaluation of Systems, International Conference on*, 0:31–40.

[29] Donatelli, S., Haddad, S., and Sproston, J. (2009). Model checking timed and stochastic properties with CSL^{TA}. *IEEE Transactions on Software Engineering*, 35(2):224–240.

[30] Emerson, E. A. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier.

[31] Emerson, E. A. and Halpern, J. Y. (1983). "sometimes" and "not never" revisited: on branching versus linear time (preliminary report). In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 127–140, New York, NY, USA. ACM.

[32] Fantechi, A., Gnesi, S., and Ristori, G. (1994). Model checking for action-based logics. *Formal Methods in System Design*, 4:187–203.

[33] Hansson, H. and Jonsson, B. (1994). A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535.

[34] Harrison, P. G. and Knottenbelt, W. J. (2002). Passage time distributions in large Markov chains. *SIGMETRICS Perform. Eval. Rev.*, 30(1):77–85.

[35] Haverkort, B. R., Cloth, L., Hermanns, H., and Katoen, J.-P. (2002). Model checking performability properties. In *DSN'02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 103–112, Washington, DC, USA. IEEE Computer Society.

[36] Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., and Siegle, M. (2000). A Markov chain model checker. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 347–362.

[37] Huth, M. R. A. and Ryan, M. D. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England.

[38] Kemeny, J. G. and Snell, J. L. (1960). *Finite Markov Chains*. D.Van Nostrand Company, Inc, Princeton, NJ.

[39] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[40] Kurshan, R. P. (1994). *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA.

[41] Kwiatkowska, M. (2007). Quantitative verification: models techniques and tools. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 449–458, New York, NY, USA. ACM.

[42] Kwiatkowska, M., Norman, G., and Parker, D. (2004). Prism 2.0: a tool for probabilistic model checking. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 322–323.

[43] Kwiatkowska, M., Norman, G., and Parker, D. (2006). Game-based abstraction for Markov decision processes. In *In Proc. of QEST: Quantitative Evaluation of Systems*, pages 157–166. IEEE Computer Society.

[44] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer.

[45] Kwiatkowska, M., Norman, G., Parker, D., and Sproston, J. (2003). Performance analysis of probabilistic timed automata using digital clocks. In *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS03), volume 2791 of LNCS*, pages 105–120. Springer.

[46] Laroussinie, F. and Sproston, J. (2005). Model checking durational probabilistic systems. In Sassone, V., editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 140–154. Springer.

[47] Liu, Y. and Trivedi, K. S. (2004). A general framework for network survivability quantification. In Buchholz, P., Lehnert, R., and Pióro, M., editors, *MMB*, pages 369–378. VDE Verlag.

[48] Lopez, G. G. I., Hermanns, H., and Katoen, J. (2001). Beyond memoryless distributions: Model checking semi-Markov chains. In *In Proceedings of the Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification, volume 2165 of LNCS*, pages 57–70. Springer-Verlag.

[49] Meyer, J. (1980). On evaluating the performability of degradable computing systems. *Computers, IEEE Transactions on*, C-29(8):720–731.

[50] Miner, A. (2000). *DATA STRUCTURES FOR THE ANALYSIS OF LARGE STRUCTURED MARKOV MODELS*. PhD dissertation, College of Williams of Mary in Virginia, Williamsburg, VA.

[51] Miner, A. S. (2004). Implicit GSPN reachability set generation using decision diagrams. *Perf. Eval.*, 56(1-4):145–165.

[52] Miner, A. S. and Jing, Y. (2010). A formal language toward the unification of model checking and performance evaluation. In *Analytical and Stochastic Modeling Techniques and Applications*, pages 130–144.

[53] Muppala, J. K., Ciardo, G., and Trivedi, K. S. (1993). Modeling using stochastic reward nets. In *MASCOTS' 93: Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 367–372. Society for Computer Simulation.

[54] Nicola, R. D. and Vaandrager, F. W. (1990). Action versus state based logics for transition systems. In *LITP Spring School on Theoretical Computer Science*, pages 407–419.

[55] Obal, II, W. D. and Sanders, W. H. (1999). State-space support for path-based reward variables. *Performance Evaluation*, 35(3-4):233–251.

[56] Pissanetzky, S. (1984). *Sparse matrix technology*. Academic Press, London, Orlando. Includes index.

[57] Pitman, J. (1993). *Probability*. Springer-Verlag, New York.

[58] Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13:45–60.

[59] Ross, S. M. (2006). *Introduction to Probability Models, Ninth Edition*. Academic Press, Inc., Orlando, FL, USA.

[60] Stewart, W. (1994). *Introduction to the Numerical Solution of Markov Chains*. Princeton Univ. Press.

[61] Suto, T., Bradley, J. T., and Knottenbelt, W. J. (2007). Performance trees: Expressiveness and quantitative semantics. In *QEST '07: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 41–50, Washington, DC, USA. IEEE Computer Society.

[62] Tewarson, R. (1973). *Sparse Matrices*. Mathematics in science and engineering : a series of monographs and textbooks. Academic Press.

[63] Trivedi, K. S., Ciardo, G., Malhotra, M., and Sahner, R. A. (1993). Dependability and performability analysis. In *Performance Evaluation of Computer and Communication Systems, Joint Tutorial Papers of Performance '93 and Sigmetrics '93*, pages 587–612, London, UK. Springer-Verlag.

[64] Vardi, M. Y. (1996). An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

[65] Vardi, M. Y. and Wolper, P. (1986). Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221.