UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

A Tool for the Evaluation of PCTL Formulas for the Verification of Markov Chains

Ahmed Mansour
Matrikelnummer: 3172131

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Abteilung Informatik und Angewandte Kognitionswissenschaft
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

July 29, 2024

Betreuer:
Prof. Dr. Barbara König
Karla Messing

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit bzw. im Fall einer Gruppenarbeit den von mir entsprechend gekennzeichneten Anteil an der Arbeit selbständig verfasst habe. Ich habe keine unzulässige Hilfe Dritter in Anspruch genommen. Zudem habe ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Ausführungen (insbesondere Zitate), die anderen Quellen wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht.

Ich versichere, dass die von mir in elektronischer Form eingereichte Version dieser Arbeit mit den eingereichten gedruckten Exemplaren übereinstimmt.

Mir ist bekannt, dass im Falle eines Täuschungsversuches die betreffende Leistung als mit "nicht ausreichend" (5.0) bewertet gilt. Zudem kann ein Täuschungsversuch als Ordnungswidrigkeit mit einer Geldbuße von bis zu 50.000 Euro geahndet werden. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuchs kann ich zudem exmatrikuliert werden.

Mir ist bekannt, dass sich die Prüferin oder der Prüfer bzw. der Prüfungsausschuss zur Feststellung der Täuschung des Einsatzes einer entsprechenden Software oder sonstiger elektronischer Hilfsmittel bedienen kann.

Ort, Datum	Unterschrift

Acknowledgements

I am deeply grateful to my supervisor, Karla Messing, for her invaluable guidance and unwavering support throughout this thesis. Her expertise and feedback have been crucial in shaping this work, and I greatly appreciate her dedication to my academic growth.

To Kwabena, Utku, and Tran, thank you for believing in me and supporting me throughout this journey. Special thanks to Kaivan for his invaluable support and for sharing his extensive knowledge, which significantly contributed to the development of this research.

To my family, my brother Mostafa Alaa, and my friends Youssef Ragab and Mohamed Taalab, your unwavering support and understanding have been my pillars of strength. Your encouragement and belief in my abilities have made this endeavor possible. I am deeply grateful for your love, patience, and constant motivation.

In conclusion, I extend my heartfelt appreciation to all those who have supported me, directly or indirectly, in completing this thesis. Your contributions have been invaluable, and I am truly fortunate to have such a supportive network.

Contents

1 Introduction

The rapid advancement of technology has integrated numerous systems into our daily lives, from household appliances such as washing machines to personal computers and satellites orbiting the Earth. These systems operate according to programs prone to human mistakes, making it essential to ensure they perform as expected through a process known as model checking.

This thesis explores stochastic process models, with a particular emphasis on Markov chains. It also examines Probabilistic Computation Tree Logic (PCTL) formulas, which are employed in our model-checking process to specify the properties we wish to verify within these models. Ensuring the reliability and performance of such systems is crucial, especially given the increasing complexity and reliance on stochastic processes across various domains. Our objective is to develop a comprehensive tool capable of rigorously analysing and verifying the probabilistic behaviour of these systems.

In this thesis, we introduce a novel tool developed in VB.NET designed to read, interpret, and evaluate PCTL formulas of Markov chain models. Our tool enables users to analyse the probabilistic behaviour of systems, ensuring that they meet the desired specifications and performance criteria. Through this work, our aim is to contribute to the field of model checking and to enhance the reliability of complex systems.

The structure of the thesis is as follows:

- **Chapter 2: Theoretical Background**: We will discuss the fundamentals of model checking and its importance, followed by an exploration of Markov chains and PCTL formulas. We will also cover the algorithms used to evaluate these formulas.
- **Chapter 3: Implementation**: We will describe the design and implementation of our tool, detailing how we parse Markov chain models and PCTL

formulas as well as evaluating them. Finally, we will discuss the integration of our tool with Python to expand its usage.

- **Chapter 4: Evaluation**: We will examine the validity and performance of our code, presenting results from various tests and case studies.
- **Chapter 5: Conclusion**: We will summarize the findings of our work and suggest potential improvements to enhance the performance and capabilities of our tool.

2 Theoretical Background

2.1 Model Checking

When designing a complex system, hardware or software, one should be able to verify these systems and make sure that they follow the desired behaviour. There are techniques to achieve this in a proper way that should ease the verification process and increase their coverage. One type of these techniques is the *Model-based* verification techniques.

2.1.1 Model-Based Verification Techniques

Model-based verification techniques rely on mathematically precise and clear models that describe the behavior of the inspected system. It turns out - before any verification— that some issues are revealed like incompleteness, ambiguities, and inconsistencies in the informal specifications of the system [2]. These issues are usually seen much later while designing. In the verification process, we use some algorithms to explore all states of the system model systematically. This sets a basis for a number of verification techniques such as exhaustive exploration which is used for model checking, which points towards the need for a smart technique for model-based verification.

2.1.2 Model Checking

Model checking is a verification technique that checks all the possible states of a system model, mostly in a brute-force way. In this way, a model checker can examine all system scenarios in a systematic manner to show that a given system model truly satisfies a certain property. For example, in a vending machine model, we might want to verify that the balance never drops below zero. Similarly, in a video game, we would verify that the player's health never falls below zero. The

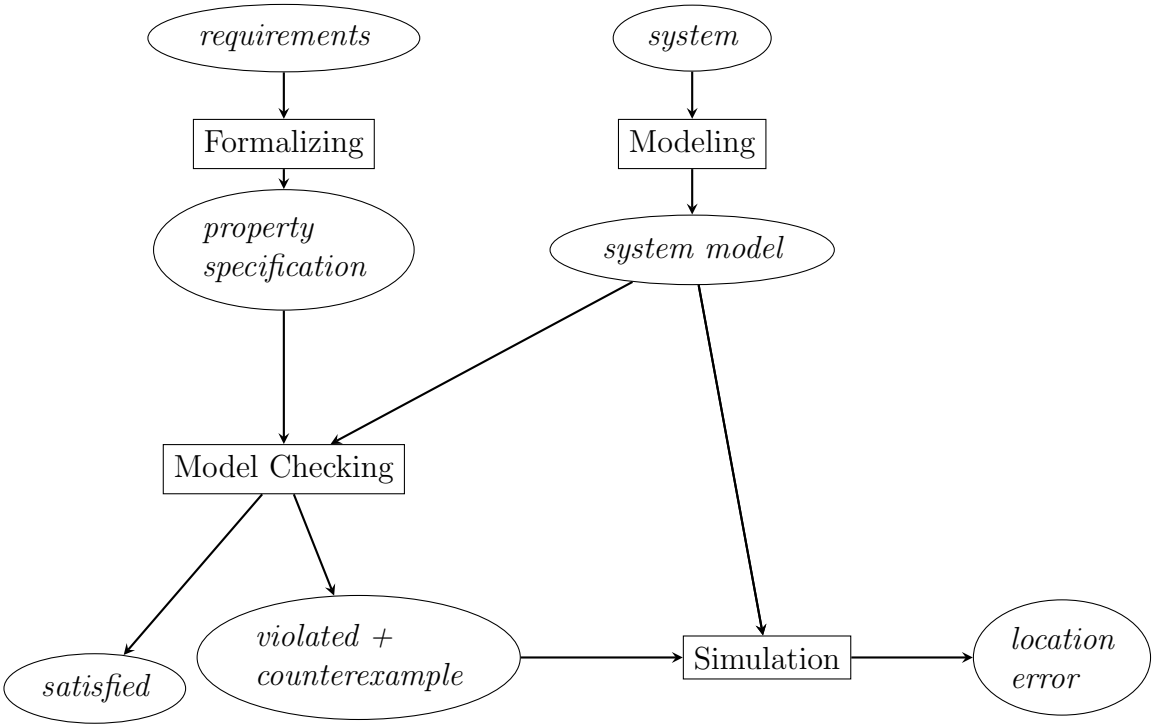

Figure 2.1: Schematic view of the model-checking approach

problem with brute-force checking is that computers of today still cannot examine every single state of a (possibly large) state space.

We see in Figure 2.1 how model-checking is performed. One can see how model checking depends on the system model as well as on the property specification. Usually, the system model is generated from a model description, which is specified in programming languages like C, and Java, or hardware description languages such as Verilog or VHDL. In addition to the system model, property specification is also required as it describes what the system should do and not do, whilst the model description shows how the system behaves. In the following sections, we will explore the model-checking process and discuss effective strategies to address it.

2.1.3 Application

Model checking has proven to be a valuable tool in various Information and Communications Technology (ICT) systems and applications. In particular, it identified five previously undetected errors within the execution module of the Deep Space 1 spacecraft controller (see Figure 2.2), as detailed in the book *Principles*

of model checking[2].

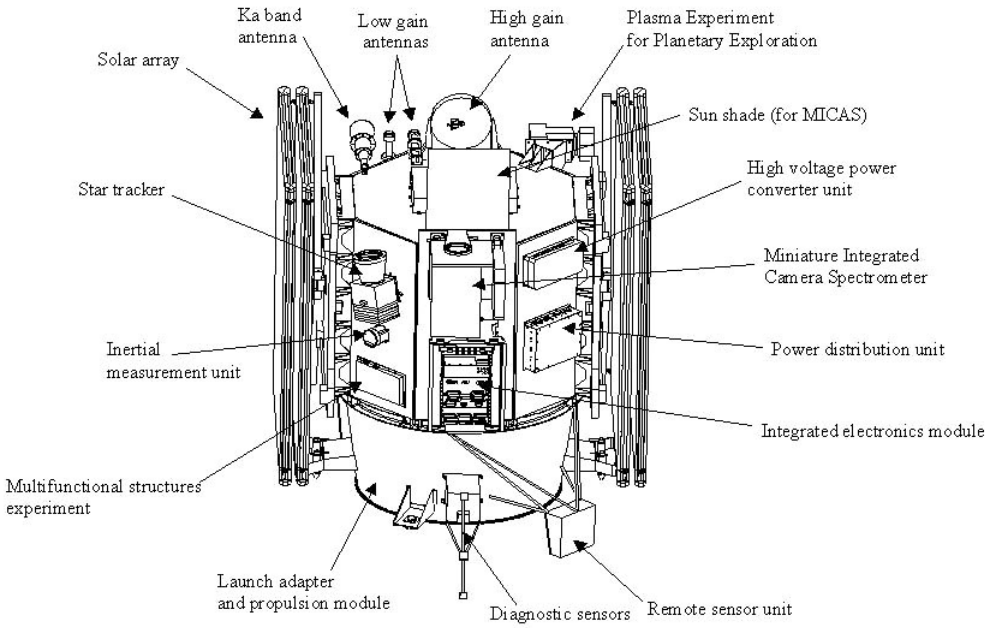

Figure 2.2: Examined modules of NASA's Deep Space-1 space-craft [10]

Most errors, similar to those discovered in the Deep Space-1 spacecraft's controller, are associated with classical concurrency issues. Unanticipated process interleavings can lead to undesirable events. This is analysed through a program where three processes—`Inc`, `Dec`, and `Reset`—collaborate. They interact with a shared variable x, which has an arbitrary initial value and is accessible and modifiable by each process. The value of x is always expected to be between 1 and 200 as stated in the book [2]. After applying model-checking techniques to this system, it's revealed that an unwanted state is reached when the value of x can be negative following a specific execution order of processes. We will show next how a system model is described using Markov chains so that we can use them in our model-checking techniques.

2.2 Markov Chains

A Markov chain is a mathematical system that is used to model stochastic processes and is visually represented by state diagrams. This system undergoes transitions from one state to another within a finite or countable state space. One variant of the Markov chain is Discrete-Time Markov Chain (DTMC) where the

transitions occur at finite time points (equally spaced). In the following parts of this thesis, DTMCs will simply be referred to as Markov chains.

2.2.1 Definition of a Markov Chain

Definition 1 *In [2], a Markov chain is described through its components as a Tuple* $\mathcal{M} = (S, \boldsymbol{P}, \iota_{init}, AP, L)$ *where:*

- S *is a countable, nonempty set of states,*
- $\boldsymbol{P} : S \times S \to [0, 1]$ *is the transition probability function such that for all states* s*:*

  $$\sum_{s' \in S} \boldsymbol{P}(s, s') = 1,$$

  and we can also describe this function as a matrix $(\boldsymbol{P}_{s,t})_{s,t \in S}$*,*
- $\iota_{init} : S \to [0, 1]$ *is the initial distribution such that:*

  $$\sum_{s \in S} \iota_{init}(s) = 1,$$

- AP *is a set of atomic propositions and* $L : S \to 2^{AP}$ *a labeling function.*

It is essential to note that Markov chains have a defining characteristic which is the Markov property *memorylessness*. This property asserts that the future state depends only on the current state and not on the states that precede the current state. One can see this property as follows:

Definition 2 *The memorylessness property states that:*

$$P(X_{t+1} = x_{t+1} \mid X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_0 = x_0) = P(X_{t+1} = x_{t+1} \mid X_t = x_t)$$

where $X_t = x_t$ *asserts that the state of the model at time* t *is* x_t*.*

2.2.2 Markov Chain Example

Now we can see the tuple $\mathcal{M}$ for a Markov chain of a simple communication protocol, which is mentioned in the book [2]. As seen in Definition 1, this tuple includes the set of states S, the transition probability matrix $\mathbf{P}$, the initial distribution state vector ι_{init}, the set of atomic propositions AP and the labeling function L. We can view the tuple $\mathcal{M}$ as follows:

$$S = \{start, try, lost, delivered\},$$

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{10} & \frac{9}{10} \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \iota_{init} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

$$AP = \{\texttt{start}, \texttt{try}, \texttt{lost}, \texttt{delivered}\},$$

$$L = \{(start, \texttt{start}), (try, \texttt{try}), (lost, \texttt{lost}), (delivered, \texttt{delivered})\}.$$

The way to understand this transition matrix $\mathbf{P}$ is that each row in the matrix belongs to one unique state in the chain here in the same order[1] as stated in S, and each element in that row represents the transition from that state to another state (or itself) also in the same order as stated in S. In other words, an element p_{ij} in the matrix represents the probability of moving from state s_i to state s_j, where i and j are the indices of row and column, respectively. It's now trivial to see that the diagonal elements p_{ii} represent the probability of self-transitions. In the same way, one can see ι_{init} as a vector where each element $\iota_{\text{init}}(s_i)$ represents the probability that the system will start in state s_i.

2.2.3 State Diagram

As seen previously, we expressed a simple communication protocol as a Markov chain. It can be visualised as an underlying digraph where each state is similar to a vertex and there is an edge from a state s to another state s' if and only if $\mathbf{P}_{s,s'} > 0$.

In Figure 2.3, we can see a Markov chain for this protocol. Here, errors could occur, which means that messages may be lost. Denoted by the arrow pointing to the state *start*, one can see that $\iota_{\text{init}}(start)$=1 and $\iota_{\text{init}}(s) = 0$ for all other states. This means that the protocol always starts at the state *start*. The diagram describes how the process works in order to send off a message along the channel. First, we

[1]Elements of a set have no inherent order; we use the written order only for illustration.

start at state *start* then we go to the state *try* where an attempt is made to send off the message, but there's a probability of $\frac{1}{10}$ that the message will be lost, in that case, another attempt is made until the message is delivered in the end. Once the message has been sent successfully, the model returns to the initial state *start* and the process can repeat in the same manner.

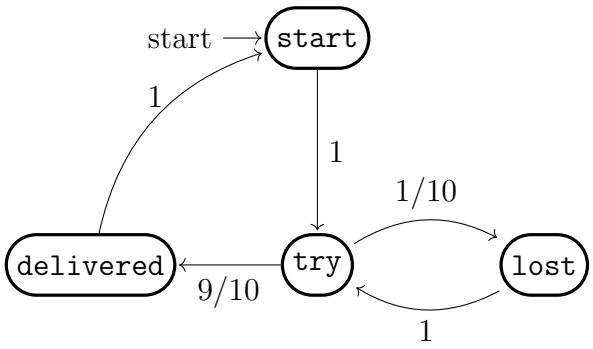

Figure 2.3: An example for a Markov chain

2.2.4 Craps Game

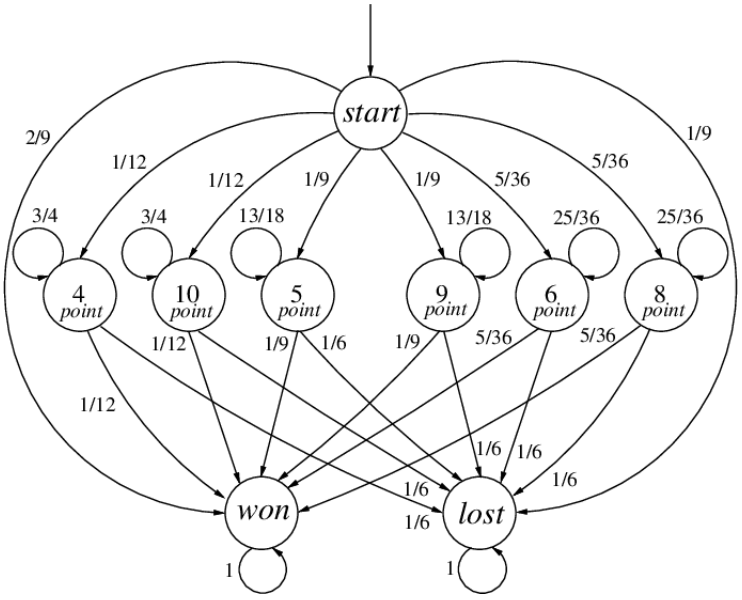

Figure 2.4: Markov chain for the behaviour of the craps game

Another model, which we will use later as an example, is the craps game described in the book [2] as a Markov chain (see Figure 2.4). In craps, players win by wagering on the outcome of two rolled dice. The initial roll, known as the "come-out roll," determines the game's course. The game-play is described as follows:

- **Winning on the Come-Out Roll**: The player wins if the sum of the dice is 7 or 11.

- **Losing on the Come-Out Roll**: If the sum is 2, 3, or 12, the player loses (called "craps").

- **Re-Rolling (Establishing a Point)**: Any other outcome (4, 5, 6, 8, 9, or 10) establishes a "point." The player continues rolling the dice until either:

  – **Winning by Matching the Point**: The sum matches the point again.

  – **Losing by Rolling a 7**: The sum is 7 before matching the point.

We can also describe it as a tuple $\mathcal{M}$:

$$S = \{start, 4, 10, 5, 9, 6, 8, won, lost\},$$

$$\mathbf{P} = \begin{pmatrix} 0 & \frac{1}{12} & \frac{1}{12} & \frac{1}{9} & \frac{1}{9} & \frac{5}{36} & \frac{5}{36} & \frac{2}{9} & \frac{1}{9} \\ 0 & \frac{3}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{12} & \frac{1}{6} \\ 0 & 0 & \frac{3}{4} & 0 & 0 & 0 & 0 & \frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 0 & \frac{13}{18} & 0 & 0 & 0 & \frac{1}{9} & \frac{1}{6} \\ 0 & 0 & 0 & 0 & \frac{13}{18} & 0 & 0 & \frac{1}{9} & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & \frac{25}{36} & 0 & \frac{5}{36} & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{25}{36} & \frac{5}{36} & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \iota_{init} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

$$AP = \{\texttt{start}, \texttt{4}, \texttt{10}, \texttt{5}, \texttt{9}, \texttt{6}, \texttt{8}, \texttt{won}, \texttt{lost}\},$$

$$L = \{(start, \texttt{start}), (4, \texttt{4}), (10, \texttt{10}), (5, \texttt{5}), (9, \texttt{9}), (6, \texttt{6}), (8, \texttt{8}), (won, \texttt{won}), (lost, \texttt{lost})\}.$$

2.3 Reachability Problems

2.3.1 Introduction

In Section 2.2.3 a simple communication protocol was visualised with a Markov chain and with a final goal to deliver the message. One can take a look at that diagram and argue about the chance of eventually delivering the message. This problem is a reachability problem, where we want to calculate the probability of the model ending up in a certain state after a bounded (or infinite) number of transitions. In the following chapters, we will show how we can solve such a problem and also the tools we need.

2.3.2 Problem Description

Let's consider the model introduced in Section 2.2.3, starting from state *start*, the message will be sent and it's expected to be successfully delivered. We need to verify that the message will always be delivered with a probability of 1. To describe that problem in the human language, we can say:

> *Starting from state **start** the system will eventually reach the state **delivered** with a probability of 1*

There are some problems with these human-language sentences like:

1. They may be misinterpreted.
2. How can a simple computer program understand them?
3. They can be long depending on the model and the property specification.

In [2], Probabilistic Computation Tree Logic (PCTL) is used to replace such sentences, which can simplify the model-checking process.

2.4 PCTL

PCTL, as defined in [2], is short for *Probabilistic computation tree logic* which is a branching-time temporal logic. PCTL formulas describe a set of states in a Markov chain. These formulas are evaluated to Boolean, that is, a state either satisfies or violates a PCTL formula.

2.4.1 Syntax

A PCTL formula includes the standard propositional logic operators and the probabilistic operator $\mathbb{P}_J(\varphi)$.

Definition 3 *PCTL state formulas are evaluated on states, and built according to the following syntax:*

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg \Phi \mid \mathbb{P}_J(\varphi),$$

where:

- Φ, Φ_1, Φ_2 are state formulas.
- $a \in AP$, and AP is the set of atomic propositions which a state satisfies as stated in Section 2.2.1.
- $J \subseteq [0, 1]$ is an interval with rational bounds.
- φ is a path formula.
- the probabilistic operator $\mathbb{P}_J(\varphi)$ means that for a state s, the probability for the set of paths satisfying φ and starting in state s is within the bounds given by J.
- $\neg$ and $\wedge$ are the *and* and *not* operators respectively of the standard propositional logic.

As we saw, the state formula with the probabilistic operator $\mathbb{P}_J(\varphi)$ has a *path formula* that is defined as follows:

Definition 4 *PCTL path formulas are evaluated on paths, and built according to the following syntax:*

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \mathbb{U} \Phi_2 \mid \Phi_1 \mathbb{U}^{\leq n} \Phi_2,$$

where:

- the operator $\bigcirc\Phi$ (pronounced "*next step*") asserts that the for the examined state s, the next state satisfies the state formula Φ.

- the operator $\Phi_1 \,\mathbb{U}\, \Phi_2$ (pronounced "*until*"), asserts for a path starting from a state s, the state formula Φ_1 will always hold along the path until the state formula Φ_2 is satisfied.

- $\Phi_1 \,\mathbb{U}^{\leq n}\, \Phi_2$ is the *step-bounded* variant of the $\Phi_1 \,\mathbb{U}\, \Phi_2$, which means that Φ_2 will hold within at most n steps.

2.4.2 PCTL Symantics

In the previous section, we showed the syntax of PCTL formulas, now let us show their semantics as described in the book [2].

Let $a \in AP$ be an atomic proposition, $\mathcal{M} = (S, P, \iota_{\text{init}}, AP, L)$ be a Markov chain, state $s \in S$, Φ, Ψ be PCTL state formulas, and φ be a PCTL path formula. The satisfaction relation $\models$ is defined for state formulas by:

$$
\begin{aligned}
&s \models a \quad \text{iff} \quad a \in L(s),\\
&s \models \neg\Phi \quad \text{iff} \quad s \not\models \Phi,\\
&s \models \Phi \wedge \Psi \quad \text{iff} \quad s \models \Phi \text{ and } s \models \Psi,\\
&s \models \mathbb{P}_J(\varphi) \quad \text{iff} \quad Pr(s \models \varphi) \in J.
\end{aligned}
$$

Here, $Pr(s \models \varphi)$ is the probability of the existence of a path starting from s where this path satisfies φ. We will explain more about paths in Section 2.5.

Given a path π in $\mathcal{M}$, the satisfaction relation is defined:

$$
\begin{aligned}
&\pi \models \bigcirc\Phi \quad \text{iff} \quad \pi[1] \models \Phi,\\
&\pi \models \Phi\mathbb{U}\Psi \quad \text{iff} \quad \exists j > 0.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi)),\\
&\pi \models \Phi\mathbb{U}^{\leq n}\Psi \quad \text{iff} \quad \exists 0 \leq j \leq n.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi)),
\end{aligned}
$$

where for path $\pi = s_0 s_1 s_2 \ldots$ and integer $i \geq 0, \pi[i]$ denotes the $(i+1)$-st state of π, i.e., $\pi[i] = s_i$.

We will use the previous semantics in the upcoming chapters to evaluate the PCTL formulas, but first, let us see how we can form correct PCTL formulas and show incorrect cases.

- **Correct PCTL state formulas**:

  - $\mathbb{P}_{[0.5,0.7]}(\mathrm{a}\,\mathbb{U}\,b)$
  - $\mathbb{P}_{[0.2,0.5]}(\mathrm{a}\,\mathbb{U}^{\leq 5}\,b)$
  - $\mathbb{P}_{=1}(\bigcirc b)$
  - $true \wedge \mathbb{P}_{=1}(\bigcirc b)$

- **Incorrect PCTL state formulas**:
  - $\bigcirc b$, as it is a path formula and exists only inside the probabilistic operator $\mathbb{P}_J$,
  - $\mathbb{P}_{=1}(true \wedge \bigcirc b)$, as $\bigcirc b$ is a path formula and can't be conjuncted with a state formula *true*,
  - $\mathbb{P}_{=1}(\mathbb{P}_{[0.2,0.5]}(true \wedge \bigcirc b))$, as $\mathbb{P}_{[0.2,0.5]}(true \wedge \bigcirc b)$ is a state formula and the outer $\mathbb{P}_{=1}$ operator is expecting a path formula.

a and b are the labels of some states.

We have explored the construction of PCTL formulas; now let us take advantage of them to articulate our objectives. Considering the problem in Section 2.3.2, the *eventually* operator $\Diamond$ can be used to describe it, [2] which can be expressed using the PCTL *until* formula $\Phi_1\,\mathbb{U}\,\Phi_2$ as follows:

$$\Diamond\,\Phi = \texttt{true}\,\mathbb{U}\,\Phi.$$

Now, the formula is written as follows:

$$\mathbb{P}_{=1}(\texttt{true}\,\mathbb{U}\,\texttt{delivered}),$$

where `delivered` is the atomic proposition of the state *delivered*.

2.5 Paths in Markov Chains

In the subsequent sections, we will delve into algorithms designed to investigate specific states within a Markov chain. In order to understand these algorithms, it is imperative to first clarify the fundamental concepts and definitions related to graphs within the context of Markov chains.

2.5.1 Definition of a Path

As discussed in Section 2.4.1, we introduce *path formulas*, which are evaluated on paths. Additionally, we showed, in Section 2.4.2, how we can interpret PCTL path formulas using paths, now, let's define what a path is. In Markov chains, a path is a sequence of states visited during the execution of the model. It can be infinite if the model is looping between some states. An example of this is the execution of the protocol in Section 2.2.3. We can describe the path $\hat{\pi}_1$ starting from state *start* and modelling only one failure and then a successful delivery as follows:

$$\hat{\pi}_1 = start\ try\ lost\ try\ delivered,$$

and a general way to represent path fragments with multiple failures, before a successful delivery, is mentioned in the book [2] as follows:

$$\hat{\pi}_n = start\ try\ (lost\ try)^n\ delivered.$$

2.5.2 Post and Pre States

Starting from some state s in our model, we can reach other different states (or the same state s). To describe them, we can use the following notion:

$$Post(s) = \left\{ s^{'} \in S \mid P(s, s^{'}) > 0 \right\},$$

where $Post(s)$ is the set of direct successors of the state s. Another set to consider is the $Post^*(s)$ set, which is the set of all states that are reachable through a finite path from the state s.

Similarly, $Pre(s)$ is the set of the direct predecessors of the state s and defined as follows:

$$Pre(s) = \left\{ s^{'} \in S \mid P(s^{'}, s) > 0 \right\}.$$

In the same way, we can define $Pre^*(s)$ as the set of states that can reach the state s through a finite path.

Applying the previous concepts to the model in Section 2.2.3, one can see that:

- $Post(start)$=$\{try\}$
- $Post^*(start) = \{start, try, lost, delivered\}$
- $Pre(delivered) = \{try\}$
- $Pre^*(delivered) = \{start, try, lost, delivered\}$

Subsequently, we will use such notation to describe our model and certain sets of states that will be helpful to solve the reachability problems.

2.6 Formulas Evaluation

As we see in the definition of a PCTL state formula in Section 2.4.1, we have 5 types of state formulas. All of them can be evaluated intuitively except for the probabilistic operator $\mathbb{P}_J(\varphi)$. In order to evaluate the probabilistic operator $\mathbb{P}_J(\varphi)$, one needs to evaluate the path formula φ. A path formula, as mentioned in Section 2.4.1, has either a *next-step* operator, a *bounded-until* operator, or an *unbounded-until* operator. We will show in the following sections how to evaluate the three types of path formulas.

2.6.1 Next-Step Formulas

As we have shown in Section 2.4.1, a *next-step* formula $\bigcirc \Phi$ is a path formula that describes the probability of the event that for some state s_1 there exists another state s_2 that satisfies the state formula Φ and is directly reachable from the state s_1.
In [2], a way to evaluate the *next-step* formula is described as follows:

$$Pr(s \models \bigcirc \Psi) = \sum_{s' \in Sat(\Psi)} \mathbf{P}(s, s'),$$

where:

- $\mathbf{P}$ is the transition probability function of $\mathcal{M}$ as seen in Definition 1.
- $Sat(\Psi)$ is the set of all states that satisfy the state formula Ψ. It is represented as a bit vector $(b_s)_{s \in S}$ where $b_s = 1$ if and only if $s \in Sat(\Psi)$.

We can then calculate $Pr(s \models \bigcirc \Psi)$ - in matrix-vector notation - by multiplying the transition probability matrix $\mathbf{P}$ with the vector for $Sat(\Psi)$, i.e. the bit vector $(b_s)_{s \in S}$.

2.6.2 Bounded Until Formulas

In Dr. Parker's PRISM lectures [11], it is described how to tackle a finite (bounded)-until-matrix $\Phi_1 \ \mathbb{U}^{\leq n} \ \Phi_2$. First, we need to identify two sets of states that trivially satisfy or don't satisfy the formula, i.e. evaluate to 1 or 0 respectively. These two sets, namely $S_{=0}$ and $S_{=1}$, can be defined as follows:

$$S_{=1} = Sat(\Phi_2),$$

where $S_{=1}$ is the set of states that satisfy the `StateFormula` Φ_2, and

$$S_{=0} = S \setminus (Sat(\Phi_1) \cup Sat(\Phi_2)),$$

where $S_{=0}$ is the set of all states that satisfy neither the state formula Φ_1 nor Φ_2.

Both sets can be seen on the Venn Diagram 2.5, where $S_{=1}$ is represented by the bigger circle and $S_{=0}$ by the area outside the 2 circles.

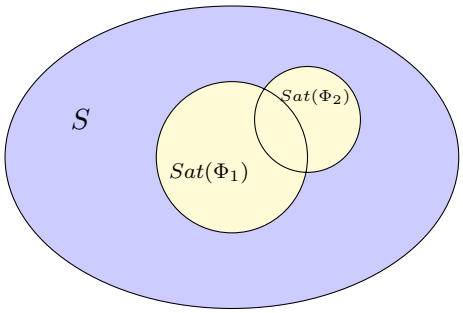

Figure 2.5: Venn Diagram to visualize $S_{=0}$

We will later see in the next chapter how we can obtain these two sets by performing searching algorithms to find the *Sat* vectors and then apply set operations.

Using these two sets $S_{=0}$ and $S_{=1}$ we can describe a third set $S_?$:

$$S_? = S \setminus (S_{=0} \cup S_{=1}),$$

which is the non-trivial set of states, and for this set we need to evaluate our path

formula. For that, we follow a least fixed point characterization as follows:

$$\mathbf{x}^{(0)} = \mathbf{0} \quad \text{and} \quad \mathbf{x}^{(n+1)} = \mathbf{A}\mathbf{x}^{(n)} + \mathbf{b} \quad \text{for n} \geq 0,$$

where:

- $\mathbf{x}$ is the probability vector which we are trying to calculate.
- $\mathbf{A}$ is the transition probability matrix for the states in $S_?$ (described as $\mathbf{A} = (\mathbf{P}(s,t))_{s,t \in S_?}$), and in each iteration it is multiplied by the probability vector $\mathbf{x}$ from the previous iteration.
- the probability vector $\mathbf{b}$ holds the probability to reach $S_{=1}$ states in one step, which is defined as $(b_s)_{s \in S_?}$ where $b_s = \mathbf{P}(s, S_{=1})$.
- $\mathbf{0}$ is the zero vector.

Using the previous least-fixed point characterisation, we can iterate over it as many iterations as the hop count. This method eventually yields the probability vector $\mathbf{x}$ which contains the probability of having a path starting from a state s and satisfying the path formula provided. This algorithm, often called *power method* [2], can also be used to evaluate the unbounded until formula by aborting the algorithm as soon as $\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)}$, however, we will next show another algorithm how to calculate the exact evaluation for it.

2.6.3 Unbounded Until Formulas

As previously mentioned, we can already calculate an approximated evaluation for the unbounded until formulas ($\Phi_1 \ \mathbb{U} \ \Phi_2$) using the *power method*. We will now discuss how to evaluate it in a more efficient way.

In [11], it is shown how we can solve linear equations to determine the probability vector $\mathbf{x}$. Similar to the bounded-until formula, we will the same definition for two sets of states $S_{=0}$, $S_?$ and find $S_{=1}$ in a different way. The following algorithm searches for all the states that aren't in the set $S_{=0}$ or don't reach any state in $S_{=0}$ through a finite path π:

Algorithm 1 $FIND\_S1(Sat(\phi_1), Sat(\phi_2), S_{=0})$

1: $R := S_{=0}$
2: $done :=$ **false**
3: **while** ($done =$ **false**) **do**
4: $\quad R' := R \cup \{s \in (Sat(\phi_1) \setminus Sat(\phi_2)) \mid \exists s' \in R.\ \mathbf{P}(s, s') > 0\}$
5: $\quad$ **if** $R' = R$ **then** $done :=$ **true**
6: $\quad R := R'$
7: **end while**
8: **return** $S \backslash R$

Now we can solve the following linear system of equations:

$$\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b},$$

where $\mathbf{A}$ and $\mathbf{b}$ are defined as in Section 2.6.2, and expanded as :

$$\begin{bmatrix} \mathrm{x}_0 \\ \mathrm{x}_1 \\ \mathrm{x}_2 \\ . \\ . \\ \mathrm{x}_n \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} & \mathbf{A}_{02} & . & . & \mathbf{A}_{0n} \\ \mathbf{A}_{10} & \mathbf{A}_{11} & \mathbf{A}_{12} & . & . & \mathbf{A}_{1n} \\ \mathbf{A}_{20} & & . & & & . \\ . & & & . & & . \\ . & & & & . & . \\ \mathbf{A}_{n0} & . & . & . & . & \mathbf{A}_{nn} \end{bmatrix} \cdot \begin{bmatrix} \mathrm{x}_0 \\ \mathrm{x}_1 \\ \mathrm{x}_2 \\ . \\ . \\ \mathrm{x}_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix}$$

after some simplification steps, we can express the system as:

$$\mathbf{x}(\mathbf{I} - \mathbf{A}) = \mathbf{b},$$

where $\mathbf{I}$ is the identity matrix of the same order as $\mathbf{A}$.

Using the Gaussian elimination algorithm, we can solve this system of equations and obtain the $\mathbf{x}$ vector which represents the evaluation of the path formula for all the states in the set $S_?$.

2.7 Tree-Like Structure

After seeing the algorithms we use to evaluate the PCTL formulas, we will now show how the order of evaluation is done in a tree-like structure. We will use the

following example:

$$P_{=1}(\bigcirc P_{[0,0.1]}(true \cup (5 \wedge P_{>0.2}(\bigcirc(won \wedge P_{=1}(\bigcirc \neg start))))))$$

We want to check if a state satisfies the previous formula but as we see, it has other nested formulas so we will follow the following structure to evaluate it:

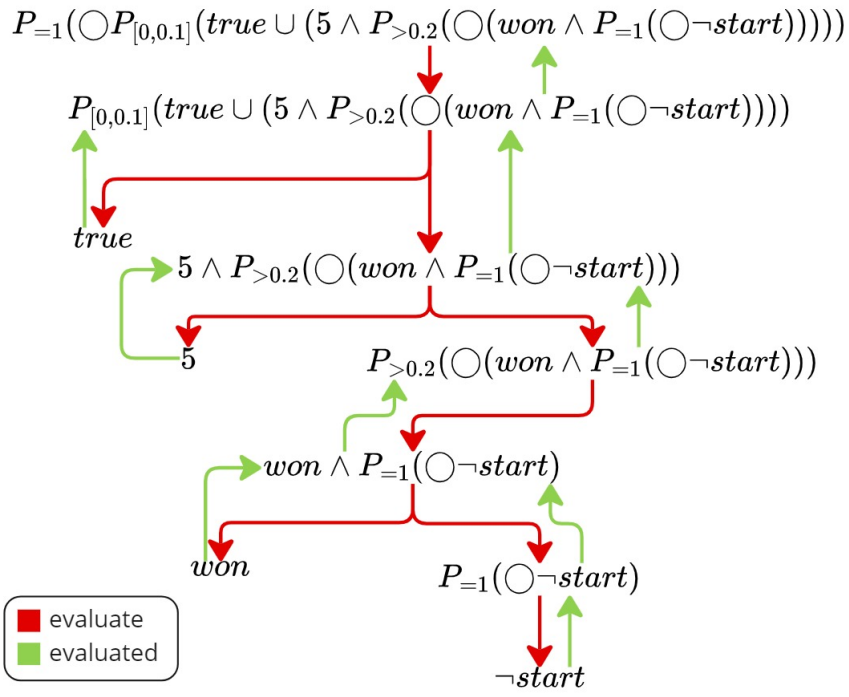

Figure 2.6: Evaluation of a State Formula in a Tree-Like Structure

In Figure 2.6, we begin by evaluating the state formula at the first level. At the second level, we encounter a nested state formula. To evaluate this second-level formula, we must consider two state formulas at the third level. One of these formulas is a Boolean formula, which is straightforward to evaluate, and the other is a conjunction state formula.

To evaluate the conjunction state formula, we examine its operands and find two state formulas at the fourth level. One of these is a label formula, which is also straightforward to evaluate, and the other is a probability state formula. The probability state formula contains a conjunction formula, which itself involves a label formula (simply evaluated) and another probability formula, as shown at the sixth level.

The probability formula at the sixth level includes a negated formula, which we evaluate by simply negating a label. After evaluating the negated formula, we return to the parent nodes at each level, sequentially evaluating them until we reach the root of the tree. This recursive process ensures that we thoroughly evaluate each component of the state formulas in the tree structure.

In Section 3.4.2, we will demonstrate how formulas are evaluated, highlighting which ones are straightforward to evaluate. However, before proceeding, we need to introduce a tool that will be used to evaluate these formulas in the following chapter.

3 Implementation

3.1 Introduction

In the preceding chapter, we delved into various foundational concepts, beginning with model checking, progressing through the structure of Markov chains, and culminating with PCTL formulas. In this chapter, we will synthesise these concepts to construct a tool with Visual Basic.NET capable of reading Markov chain models and PCTL formulas as well as evaluating these formulas. We will also show how we can test our code and visualise some of the results.

3.1.1 VB.NET

In [8], Visual Basic .NET (VB.NET) is described as an object-oriented programming (OOP) language developed and maintained by Microsoft. It is part of the .NET framework and allows for the creation of both form-based and web-based applications. In order to develop our tool in VB.NET, we used Visual Studio (2022), Microsoft's comprehensive Integrated Development Environment (IDE).

The advantage of VB.NET lies in its simplicity and ease of use, particularly for those with prior experience in an OOP language such as *Java*.

In the following sections, we will demonstrate the use of VB.NET for the development of a Command Line Interface (CLI) application we created on GitHub[1]. Then we will show how we implemented a `Core` library to read and evaluate our models. Finally, we will discuss how we can integrate our application with the programming language *Python*.

[1] `https://github.com/Mesame7/PCTL_Solver`

3.2 Console Application

3.2.1 Definition

A console application typically interacts with the user through a text-based interface. For example, it may prompt the user to enter data or select options, and then process that input to produce and display results. Unlike graphical user interfaces (GUIs), console applications do not require the overhead of managing graphical elements, making them efficient for tasks that can be performed without the need for visual aids.

```
$ Please enter a command : open my_Model.txt
$ Model created with 9 states.
$ Please enter a command : eval my_Model_PCTL_Formulas.txt
$ Formula was evaluated in 4.4ms
$ The Formula  start : ( Pin[1, 1]{ ( true ) U
( delivered ) } ) evaluates to true.
```

Code 3.1: A snapshot of the console application interaction

As demonstrated in the previous snapshot, a user can load a model from a `.txt` file and evaluate the PCTL formulas. This process is facilitated through the execution of designated commands. Next, we will demonstrate the available commands and explain how to use them.

3.2.2 Implementation

Initially, a prompt is displayed, inviting the user to enter a command. Subsequently, the input is acquired utilising the `Console.ReadLine()` method, as elucidated in [6]. Upon retrieval of the user's input, a decision is made regarding the command issued by the user, employing a `Select Case` statement.

The user has the following options to proceed with, each with specific rules:

- The `open` command allows the user to load a model into the program by specifying a `.txt` file containing the model.

- The `eval` command enables the user to load and evaluate PCTL formulas from a `.txt` file, subsequently displaying the results of the evaluation. It should always be executed after executing an `open` command to have a model for which the formulas can be evaluated.

- The `clear` command clears all loaded models and formulas, effectively resetting the program to its initial state. It can be called at any time with no dependency on any other commands.

- The `time` command can be used to toggle the display of execution time after evaluating a formula. It can also be called at any time with no dependency on any other commands.

- The `value` command can be used to toggle the display of the evaluated value of the top-level `ProbabilityFormula`. This command can be invoked at any time, independently of other commands.

- The `round` command can be used to set the number of digits to which final values of the `PathFormula` will be rounded. This is because a `Double` value in .NET has a precision between 15-17 bits as mentioned on the Microsoft Learn page [7].

- The `help` command displays descriptions of the previous commands along with examples of how to use them.

- The `exit` command closes the program.

After entering one of these commands with the appropriate arguments, the pro-

gram initiates a new process based on the specified command. The user will then receive an output indicating the state of execution for the chosen command, as we will see in the next section.

3.3 Model Parsing

In this section, we will demonstrate how to read the Markov chain model and the formulas file as text files, which the user can manually create. Additionally, we will explain their structure.

3.3.1 Text Files

To facilitate the loading of models and formulas into the program, we employ text files (`.txt`) to delineate the model structure and the associated formulas. These `.txt` files adhere to a predefined syntax and set of rules, which we will explain in detail in Section 3.3.2, intentionally simplified to enable user-friendly creation of model files.

3.3.2 Model Files Syntax

The initial type of file to be loaded is the model file. A model file encompasses details regarding each state within the Markov chain, as well as all transitions to other states. The following is the file content for the model depicted in Figure 2.3:

```
start : 1 : start : try-1
try : 0 : try : lost-1/10 , delivered-9/10
lost : 0 : lost : try-1
delivered : 0 : delivered : start-1
```

As we can see from the text, we have four lines representing four states. Each line consists of four parts separated by the colon character ":". These four parts are as follows:

1. The name of the state.

2. The initial probability of the state $\iota_{\text{init}}(s)$.

3. The atomic propositions (labels) that hold for that state separated by a comma (',').

4. The transitions originating from the state separated by a comma (',') and each transition text holds the name of the next state as well as the transition probability in a decimal format (`0.99`) or a ratio (`99\100`).

The third and fourth parts can have as many components as the system requires. These components are separated by a comma (','). For the third part, each component represents a label on that state. The components of the fourth part represent transitions out of the state. Each transition component consists of two parts: the first is the name of the next state in that transition, and the second part is the probability of this transition.

So, for the text provided above, we can interpret, for example, the second line:

```
try : 0 : try : lost-1/10 , delivered-9/10
```

into 4 different parts:

1. `try` which represents the name of the state.

2. `0` the initial probability of the *try* state.

3. `try` the atomic proposition satisfied by the state *try*

4. `lost-1/10 , delivered-9/10` can be split into the following transitions:

   - `lost - 1/10` represents a transition from the state *try* to the state *lost* with a probability of $\frac{1}{10}$

   - `delivered - 9/10` represents a transition from the state *try* to the state *delivered* with a probability of $\frac{9}{10}$

3.3.3 Formulas File Syntax

The other type of file to be loaded is the one that contains the formulas. Each formula file contains at least one PCTL formula as well as the starting state. Below is the content for the formulas file which was used in Snapshot 3.1:

```
start : ( Pin[1, 1]{ ( true ) U ( delivered ) } )
```

As we see the formula text is split into two parts by a colon (':'), the first part represents the name of the state for which the state formula is evaluated. The second part represents the formula itself with the following rules:

- the probabilistic operator $P_J(\varphi)$ is represented as `Pin[a, b]` where a and b are the interval bounds, also the bounds could be closed or opened as follows:
  - `Pin[a, b]`
  - `Pin]a, b]`
  - `Pin[a, b[`
  - `Pin]a, b[`

  and if `a` is equal to `b`, with closed brackets, then this means that we are checking if the probability is exactly equal to `a` ($P_{=a}(\varphi)$).

  In any case, the second square bracket should be followed by curly braces $\{\varphi\}$ holding a path formula as follows: `Pin[a, b]{}`

- Labels(Atomic Proposition) and boolean values should be surrounded by parentheses as follows: `( true )`, `( delivered )` respectively.

- Any conjunction can happen between 2 state formulas or more provided that parentheses surround the state formulas as well as the whole conjunction as follows: `( ( true ) ^ ( delivered ) ^ ( !  ( delivered ) ) )`, the conjunction symbol is the also known as Caret - circumflex and has the ASCI code of 0x5E.

- Any negation can be represented with an exclamation mark, provided that the negated state formula is surrounded by parentheses, as is the entire negated formula, as follows: `( !  ( delivered ) )`.

- The Next-Step operator can be written as: `X ( delivered )`.

- The Unbounded-Until operator is placed between 2 state formulas surrounded by parentheses in the following way: `( true ) U ( delivered )`.

- The Bounded-Until operator is placed similarly to the unbounded until operator with a *less than* sign ("<") followed by an equal sign ("=") and an

`Integer` representing the number of steps as follows: `( true ) U<=1000 ( delivered )`.

- It is recommended to add spaces before and after the parentheses as follows: `( ! ( delivered ) )`.

3.3.4 Parsing Files

After seeing how the files are structured, we can parse them line by line to extract the model data and formulas. Subsequently, we will see how the code is implemented for parsing both the model and formulas.

3.3.4.1 Model files

To parse the model file, we need to read and process each line as a state. After reading a line as a `String`, we split it by the colon character (:) using the `String.Split(":"c)` function. This gives us the four parts as described in Section 3.3.3.

Once all lines have been processed, we can create states using the first three parts (the name, the initial probability, and the labels). Since a state can have multiple labels, we split the part of the labels using the `String.Split(","c)` function to add all the labels to their respective states.

After compiling a list of all states, we create the transitions using the fourth part of each line. For each state, we split the last part by the comma character (","), resulting in a list of transitions as `List<String>`, each transition (`String`) consists of the to-state name as `String` and the probability of this transition, from the parsed state to the to-state, also as a `String`. The previous probability can be in the decimal or ratio format. In order to parse the decimal format, we simply use `Double.Parse()`. As for the ratio format, we use the `NCalc` library[3] with the function `Expression.evaluate()` to get the `Double` equivalent of this ratio. We then iterate over this list to create the transitions, associating each with its to-state and probability, and finally add them to the corresponding state.

3.3.4.2 Formulas File

Now we need to read the formulas file, which will be more complex than reading the model file. In this section, we use the following state formula, starting from state *start*, from the craps game (in Section 2.2.4) as a general example:

$$\mathbb{P}_{[0.0.5]}(C\ \mathrm{U}^{\leq 2} B)$$

where $C = \{\text{start}, 4, 5, 6\}$ and $B = \{\text{won}\}$. Using the PCTL state formula syntax from Section 2.4.1, it can be written as:

$$\mathbb{P}_{[0.0.5]}(\neg(\neg\text{start} \wedge \neg 4 \wedge \neg 5 \wedge \neg 6)\ \mathrm{U}^{\leq 2}\text{won})$$

According to the rules in Section 3.3.3, this state formula translates to:

```
start :( Pin[0,0.5]{ ( ! ( ( ! ( start ) ) ^
( ! ( 4 ) ) ^ ( ! ( 5 ) ) ^ ( ! ( 6 ) ) ) ) U<=2 ( won ) } )
```

This example covers most of the file syntax rules. To parse it, we start by splitting the text at the colon character (":"). The first part, `start`, indicates the state for which we are evaluating the state formula. The remainder is as follows:

```
( Pin[0,0.5]{ ( ! ( ( ! ( start ) ) ^
( ! ( 4 ) ) ^ ( ! ( 5 ) ) ^ ( ! ( 6 ) ) ) ) U<=2 ( won ) } )
```

First, we trim it to remove the leading and trailing white spaces and parentheses, using the `String.Substring()` method, resulting in the text seen in the root level of the tree in Figure 3.1. Next, we will use the same figure to explain how we parse the formula in a tree-like structure:

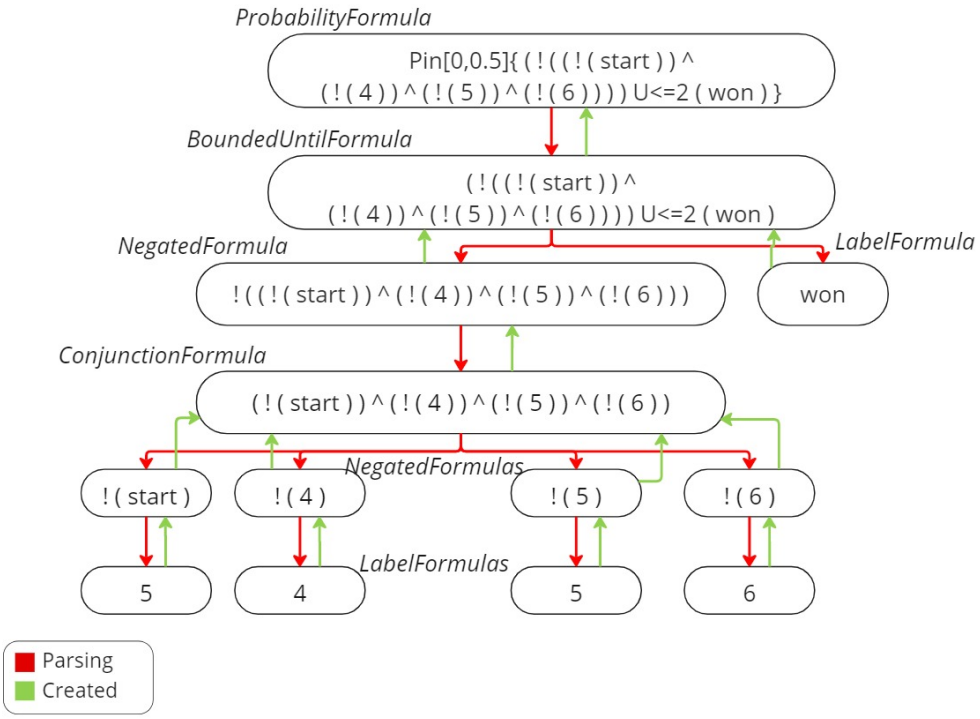

Figure 3.1: Parsing a PCTL Formula as text in a Tree-Like Structure.

First, we extract the substring `Pin[0,0.5]`, using the method `String.Substring()` for `Pin`, and then taking the `String` between the square brackets to define the probability range J as an interval taking into account the closed and open brackets. We also check if the limits are from 0 to 1, and if the lower limit is less than or equal to the upper limit. Removing the leading and trailing curly brackets ({ }), the remaining text appears as seen in the second level of the tree.

We proceed by determining the type of the `PathFormula` through a search for specific substrings (`U<=`, `U`, or `X`) in the same level of parsing (not within parentheses). To achieve this, we utilise a Regular Expression (regex), as described in MDN web docs [5] to search for the previous substrings as a pattern of chars with the `Regex.IsMatch()` function.

We then conclude that the preceding part is a `BoundedUntilFormula`, which comprises two `StateFormula`s as operands as we see in the third level of the tree.

For any `StateFormula`, we verify if it includes a negation symbol (!), a conjunction symbol (^), or a probability operator (`Pin`) in the same level of parsing (not within any parentheses). To accomplish this, we again use regex to search for these

patterns.

We apply the previous searching rule to the second part `(won)` and see that it does not contain any of the previous elements. We then figure out that it is either a `BooleanFormula` or a `LabelFormula`, and by checking if it equals either of the `String`s: `true` or `false`, we find out that it's a `LabelFormula`. Finally, we create a `LabelFormula` and stop this branch of searching, as a `LabelFormula` shouldn't contain any children nodes of `StateFormula`s or `PathFormula`s.

For the first part (left-side node in the 3rd level), we search for any of the previously mentioned elements and find that it contains only a negation (!) in the same level of parsing, we then create a `NegatedFormula` which should have a child as a `StateFormula`. After that, we take the substring inside the parentheses and parse it as we see in the fourth level of the tree.

We search again and find conjunction operators (`^`) at the same level of parsing, concluding that there is a conjunction between some `StateFormula`s. We then branch the search into 4 parts each representing a `StateFormula`: `( ! ( 4 ) )`, `( ! ( 5 ) )`, `( ! ( 6 ) )` and `( ! ( start ) )` as seen in the fifth level of the tree.

For each of these four parts we check again, according to the previously mentioned rule, to find out that they are `NegatedFormula`s, which have a `StateFormula` inside, then by going one level deeper in each part and checking again, we realize that they are `LabelFormula`s. Finally, we create the `LabelFormula`s and stop parsing at each search branch.

Finally, if the user does not follow the syntax described in Section 3.3.3, they will receive an error message on the console indicating that the syntax is incorrect. However, the message will not specify why the error occurred.

Next, we will demonstrate how we designed our program to accommodate the earlier files containing the model and the formulas.

3.4 Class Design

3.4.1 Model

In our approach, we parse two files containing the model and the PCTL formulas respectively. Subsequently, we represent these concepts as objects using VB.NET's object-oriented programming (OOP).

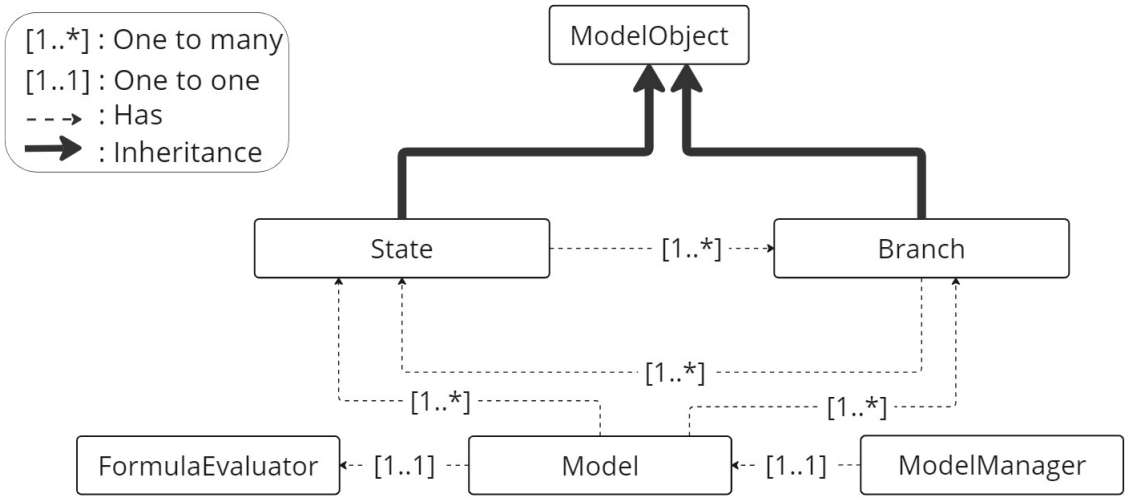

Figure 3.2: Simple Class Diagram of the Model's Structure.

As we see in Figure 3.4, the major classes of our model are as follows:

- the class `SystemManager`, which handles the creation of the model and its components like the states and branches (i.e. transitions). Also, it has an instance of the model we are evaluating.

- class `Model` which contains the model structure of states and branches. It also has an instance of `FormulaEvaluator`.

- `FormulaEvaluator` is concerned with evaluating formulas by performing network-traversing algorithms.

- `ModelObject` is the base class for `Branch` and `State`.

- `State` is the class representing a node in the Markov chain, it holds different attributes such as:

  – A list of `Label` which the states satisfies.

  - A list of `Branch`, representing the transitions out of that `state`.
  - A name
  - An initial probability as `Double` (64-bit floating point number).
  - An `Integer` id number (starting from 0) which is used as an index.

- `Branch` which represents a transition in a Markov chain and has the following attributes:
  - two `State` objects as a from-state and a to-state.
  - the probability of that transition as `Double`.

3.4.2 Formulas

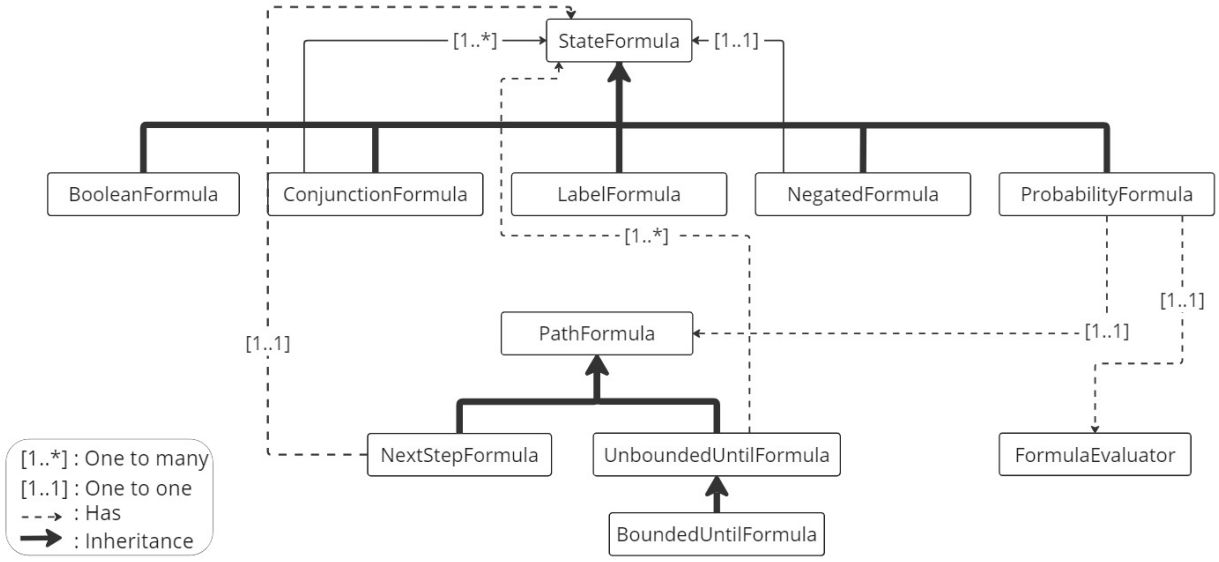

Figure 3.3: Simple Class Diagram of the Formulas' Structure.

Figure 3.3 shows the structure of the formulas and a glance of how they are related as follows:

- `StateFormula` is the base class for these 5 types of formulas, with an `Overridable` function (`Evaluate`), which all evaluate to a `Boolean` value:

  - `BooleanFormula` represents a simple boolean value which evaluates to `true` or `false`.
  - `ConjunctionFormula` is the conjunction between two or more `StateFormula` objects, this implementation is different than the syntax of PCTL formulas in Section 2.4.1 to reduce the number of parentheses, making the formulas files more user-friendly to create.
  - `LabelFormula` has a `Label` attribute.
  - `NegatedFormula` has a `StateFormula` attribute.
  - `ProbabilityFormula` has a `PathFormula` attribute, a `FormulaEvaluator` dependancy and a probability range from 0 to 1, we will later discuss how it is evaluated.

- `PathFormula` is the base class for 2 types of path formulas which are:
  - `NextFormula` that has a `StateFormula` attribute.
  - `UnboundedUntilFormula` which has 2 `StateFormula` attributes representing the 2 parts of an *until* formula and is the base class for:
    * `BoundedUntilFormula` which has one attribute, the hop count n.

Finally, we can see how this is a tree-like structure as some `StateFormula`s, such as the `ConjunctionFormula` or the `ProbabilityFormula`, have other `StateFormula`s or `PathFormula`s, respectively, as children, but first we need to evaluate the base-case `StateFormula`s to be able to evaluate their parent formulas. We will later explain, in detail, this tree-like structure and use it to evaluate the state formulas.

3.5 Formulas Evaluation

3.5.1 StateFormula

In Section 2.7, we see how the logic is evaluated in a tree-like structure. We will now see how state formulas are evaluated for a state s:

- **Base Cases:**

  - `LabelFormula`: Evaluated as `true` if the state has the label it represents.
  - `BooleanFormula`: Evaluated according to the `boolean` value it holds.

- `NegatedFormula`: Evaluated as the negation of the evaluated `boolean` value of the nested `StateFormula` it holds.
- `ConjunctionStateFormula`: Evaluated as `true` only if both the child `StateFormulas` are satisfied by the state.
- `ProbabilityFormula` (discussed in detail next): Evaluated as `true` if, starting from that state, there exists a path where the `PathFormula` evaluates to a `Double` value within the specified range.

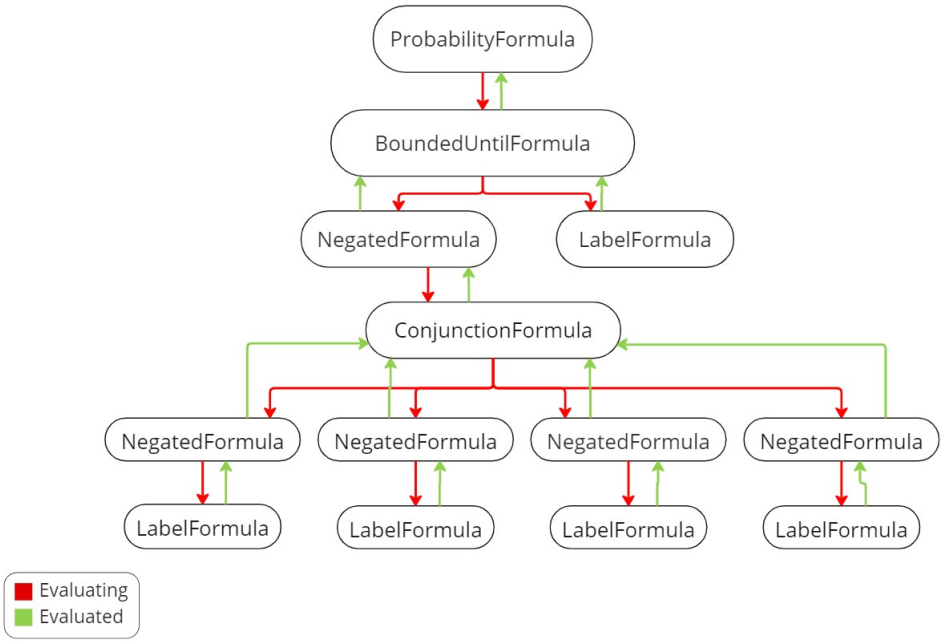

Figure 3.4: Evaluation of a PCTL Formula in a Tree-Like Structure.

In the previous figure, we illustrate the evaluation of the `StateFormula`, which is parsed in Section 3.3.4.2, within a tree-like structure. We begin at the `StateFormula`, which serves as the root of the tree. From there, we encounter other `StateFormulas` or `PathFormulas` containing state formulas as the leaves. We then continue traversing the tree until we reach a base case such as a `LabelFormula` or a `BooleanFormula`. Finally, we can start to evaluate the base cases and go up the tree evaluating until

we reach the root node. The following sections will show how we evaluate the `PathFormula`s.

3.5.2 PathFormula

Any `ProbabilityFormula`, as depicted in the class diagram in Section 3.4.2, consists of a probability interval and at least one `PathFormula`. Evaluating a `ProbabilityFormula` involves checking for the existence of a path, starting from a state s, that satisfies the `PathFormula` with a probability in the specified interval. It is important to note that a path formula may evaluate to zero, indicating no such path, that satisfies the `PathFormula`, exists.

3.5.2.1 Next Formulas

As demonstrated in Section 2.6.1, a `NextFormula` is a path formula that determines the probability of transitioning from a state s_1 to a state s_2, where s_2 satisfies a `StateFormula` Ψ. We have also detailed the mathematical evaluation of the `NextFormula` using the following notation:

$$Pr(s \models \bigcirc \Psi) = \sum_{s' \in \mathrm{Sat}(\Psi)} \mathbf{P}(s, s') = (\mathbf{P} \cdot \mathrm{Sat}(\Psi))_s \tag{3.1}$$

Next, we need to compute the $\mathrm{Sat}(\Psi)$ vector as we will use it in this section and the following sections as well. We begin by creating a zero vector of size equal to the number of states in the model. Then, we iterate over all states in the model, checking whether each state satisfies the `StateFormula` Ψ. The evaluation of the `StateFormula` for all states follows the method described in this chapter, which potentially results in a tree of `StateFormula`s or `PathFormula`s until a base case `StateFormula` is reached then we evaluate these base cases and keep going up the tree until reaching the root `StateFormula`, as outlined in Section 3.5.1.

After obtaining the $\mathrm{Sat}(\Psi)$ vector, we can implement a function to multiply a matrix by a vector. Specifically, we will multiply the transition probability matrix $\mathbf{P}$ by the $\mathrm{Sat}(\Psi)$ vector. This operation yields a vector containing the evaluations of the `NextFormula` for all states in the model. However, since we are only interested in the evaluation for the state s, we only need the value at the index corresponding to s in this resultant vector.

As we can see, the evaluation of the `NextFormula` was simple with no need for algorithms such as the evaluation of the remaining two `PathFormula`s.

3.5.2.2 Bounded Until Formulas

In Section 2.6.2, we showed how a `BoundedUntilFormula` ($\Phi_1\ \mathbb{U}^{\leq n}\ \Phi_2$) is mathematically evaluated with a least fixed point characterisation. First, we need to determine the three sets of states $S_{=0}$, $S_{=1}$ and $S_?$.

As mentioned in Dr. Parker's PRISM lectures [11], we can find $S_{=1}$ as the the set of states satisfying the `StateFormula` Φ_2:

$$S_{=1} = Sat(\Phi_2),$$

in the same way, for finding the Sat vector, described in Section 3.5.2.1.

Then we get the set $S_{=0}$ as follows:

$$S_{=0} = S \setminus (Sat(\Phi_1) \cup Sat(\Phi_2))$$

where $\setminus$ and $\cup$ are the difference and union operators, respectively. As we see, we need to find both $Sat(\Phi_1)$ and $Sat(\Phi_2)$ as lists, perform the union operator on them and finally get all the states in S which don't exist in the union set. Using *.NET*'s pre-defined functions for the `List` class, we can perform the previous operations.

After finding $S_{=0}$ and $S_{=1}$, we can simply check if the examined state, at which we are evaluating the `PathFormula`, is a member of $S_{=0}$, then we conclude that the `PathFormula` evaluates to 0 at that state. Similarly, we check if that state is a member of $S_{=1}$, then we know that `PathFormula` evaluates to 1 at that state. This rule will apply also to the `UnboudedUntilFormula` in the next section.

We can then get $S_?$ by making a copy of the S, all the states in the model, and excluding the sets $S_{=0}$ and $S_{=1}$ from it.

Having all the sets of states ready, we can finally implement a function to perform the least fixed point characterisation as described in Section 2.6.2:

$$\mathbf{x}^{(0)} = \mathbf{0} \quad \text{and} \quad \mathbf{x}^{(n+1)} = \mathbf{A}\mathbf{x}^{(n)} + \mathbf{b} \quad \text{for n} \geq 0$$

As seen in Section 2.6.2, we now create a matrix $\mathbf{A}$, which is a reduced copy of the $\mathbf{P}$ matrix, containing only the transition probabilities between the states in $S_?$.

Then we use the function mentioned earlier in Section 3.5.2.1 to multiply the matrix $\mathbf{A}$ with the $\mathbf{x}$ vector, and then use an implemented function to add matrices so that we can add the terms $\mathbf{A}\mathbf{x}^{(n)}$ and $\mathbf{b}$.

After that, we can wrap the previous operation in a function that we can call as many times as the hop count n. Finally, after the iterations, we obtain the final value of the $\mathbf{x}$ vector which contains the probabilities in the same manner as in Section 3.5.2.1, from which we can extract the corresponding value for the examined state.

3.5.2.3 Unbounded Until Formulas

As previously mentioned, we can already calculate an approximate evaluation for the `UnboundedUntilFormula`s using *power method*. We will now discuss how to evaluate it in a more efficient way.

In [11], it is shown how we can solve linear equations to determine the probability vector $\mathbf{x}$. Similar to the `BoundedUntilFormula`, we will use the same definition for two sets of states $S_{=0}$, $S_?$ and define $S_{=1}$ in a different way using the algorithm mentioned in Section 2.6.3.

Now we can solve the following linear system of equations:

$$\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$$

After some simplification steps, we can express the system as:

$$\mathbf{x}(\mathbf{I} - \mathbf{A}) = \mathbf{b},$$

where $\mathbf{I}$ is the identity matrix of the same order as $\mathbf{A}$.

In order to solve this system of equations, and obtain the $\mathbf{x}$ vector which represents the evaluation of the `PathFormula` for all the states in the $S_?$ set, we implemented a Gaussian elimination algorithm with the help of a *geeksforgeeks* Article [14]. This algorithm combines the coefficient matrix $(\mathbf{I} - \mathbf{A})$ and constants vector $\mathbf{b}$ into an augmented matrix. We then transform the matrix into an upper triangular

form by eliminating elements below the pivots. Finally, we solve for each variable starting from the last row upwards and have the resulting vector which contains the solutions to the system of equations. This vector contains the evaluations of the `PathFormula` for all the states in $S_?$, and from that vector we only need the one corresponding to the examined state.

After calculating the probability for the `PathFormula`, we then check if it falls in the range of its parent `ProbabilityFormula` to evaluate it to a `Boolean` value as mentioned in Section 3.4.2. Next, we will demonstrate how we ensure the correctness of our program during development.

3.6 Integration Tests

In this section, we discuss the integration tests implemented to ensure the correctness and reliability of the probabilistic model-checking functionality provided by the `PCTL_Solver_Core` library.

The integration tests are organized within a dedicated test class. The purpose of these tests is to validate the program's ability to read model specifications, evaluate probabilistic formulas, and compare the results against expected values. Each test follows a structured approach that involves setting up a consistent testing environment, and the execution of specific validation scenarios.

3.6.1 Setup Method

A setup method is executed before each test case to configure the testing environment. This ensures consistent formatting and parsing throughout the tests.

3.6.2 Test Cases

So far we have written three test cases. The first test reads a model file and a formula file (containing a next-step formula), evaluates the formula, and asserts that the output matches the expected value.

The second test evaluates the system's handling of a model representing a simplified version of the craps dice game. The test ensures that the evaluation result of the formula, containing a `BoundedUntilFormula`, is accurate and falls within

a very small error margin, as we sometimes round up to the nearest 6^{th} digit to avoid the big number of the actual digits.

The third test verifies the program's outcome on a basic probabilistic model with a formula file containing an `UnboundedUntilFormula`. This test checks that the evaluation produces a result of exactly 1 which is the correct evaluation for the formula.

3.6.3 Test Driven Development

These integration tests play a crucial role in ensuring the robustness and accuracy of the probabilistic model-checking functions within the `PCTL_Solver_Core` library. By systematically validating different models and their corresponding formula evaluations, the tests help to identify and rectify potential issues every time we modify the code which is called *Test Driven Development*(TDD).

3.7 Python Integration

Python is a powerful programming language that can be helpful in data analysis and manipulation. There are many libraries that we can use to describe our Markov chains and visualise them. In the following sections, we will show how we integrate our code with *Python* to expand its usage and also visualize the solving process.

3.7.1 Dynamic-link library

In MS Windows documentation [9], dynamic link library (DLL) is defined as a library that contains code and data that can be used by more than one program at the same time.

In our repository, we have four main projects:

- `Core`, which contains all the logic and objects' structure.
- `CLI`, which displays a command line interface that uses `Core` functions to load the model and solve it.

- `Tests`, where we implement integration tests with actual models to verify the correctness of the evaluation algorithms regardless of the changes we make to `Core`.

- `Export`, where we export some functions that we can use later in Python.

The `Export` project is now our main concern, and we export that project as a `dll` file so that we can import it into Python code and use the implemented functions in Python. After building our .NET project, we can find the `dll` file in the `"Export\bin"` folder and then use the Python.NET library to load it, as we will see next.

3.7.2 Importing DLL in Python

Python.NET [13] is a package that enables Python users to integrate with the .NET 4.0+ Common Language Runtime on Windows and the Mono runtime on Linux. One of its advantages is the ability to import .NET `dll` libraries into Python code, as demonstrated below.

Listing 3.1: Importing a DLL File in Python

```
import clr
def load_pctl_export_dll():
    clr.AddReference(r"PATH\TO\PCTL_Export_VB.dll")
    from PCTL_Export_VB import APIExporter
    return APIExporter.APIExport
```

This allows us to integrate our `Export` project into a Python script. Subsequently, we can call various functions such as loading models, evaluating formulas, or retrieving specific sets of states (e.g., $S_{=1}$), which are crucial for model evaluation. The outputs from these functions can be used to visualize the model or illustrate specific evaluation steps.

3.7.3 Python Libraries

In addition to utilizing our `dll` file as a library, we can utilize the `NetworkX` library [4] for visualizing our model. We can also use the `imageio` library to generate images or GIFs (series of images) that represent the model, highlighting special

sets of states ($S_{=1}$, $S_{=0}$, or $S_?$) in different colours from the rest of the network. As we see in Figure 3.5, we use the previous library to highlight the sets $S_{=1}$ and $S_{=0}$ that we use to evaluate the formula $\mathbb{P}_{=1}(\texttt{true}\,\mathbb{U}\,\texttt{delivered})$ in Section 2.4.2. These visualisations serve to verify our evaluation algorithms and help users understand their execution.

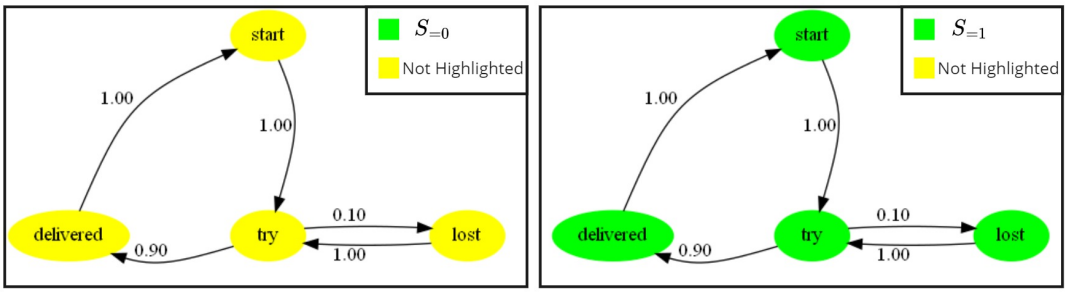

Figure 3.5: Highlighting Special States from Formula Evaluation in Section 2.4.2.

Furthermore, we can dynamically create models by writing the states, according to the rules in Section 3.3.2, to a text file using the class `FileIO` and the function `TextIOBase.write()` in Python. One can easily do this when there is a pattern of transitions that repeats over the states, as we will see later in Section 4.3.1. We will use this method to create large models, to assess our program's performance. We can then visualize the execution times on plots, demonstrating the computational complexity of our program.

4 Evaluation

As we saw in the previous chapters, we implemented a program capable of reading Markov models and checking them with PCTL formulas. In this chapter, we will show the advantages of our program as well as compare the results to an existing model checker *PRISM* to show that our program checks the models correctly.

Probilistic Symbolic Model Checker (PRISM) [12] analyses probabilistic systems such as discrete-time Markov chains (DTMC) and continuous-time Markov chains (CTMC). Similar to our program, it uses model files and formula files. In the following section, we will show how the results of our program agree with PRISM and also show the performance of our tool.

4.1 Advantages and Features

One primary advantage we focused on during the development of our program is its user-friendliness. From creating model files and formula files to executing CLI commands and utilising our custom Python library, users can effortlessly generate both model and formula files. This is made possible by following the guidelines outlined in Section 3.3.2 and Section 3.3.3 respectively.

Additionally, users can execute the commands in Section 3.2.2 to open the model, evaluate the formulas, get help with the commands, and many other helpful features.

Finally, our library can be imported into Python to allow visualization of the model, dynamic creation of large models, or perform extra mathematical analysis of the results.

4.2 Sample Models

In order to verify our program, we will use three different models: the communication protocol, the craps game, described in Section 2.2.3 and Section 2.2.4, and a birth-death model described in Article [1].

4.2.1 Communication Protocol

For the communication protocol, we evaluate the following state formulas at state `start` (see Figure 2.3):

- $\mathbb{P}_{>0.9}(\text{true}\,\mathbb{U}^{\leq 5}\,delivered)$, and it evaluates to true as the path formula evaluates to 0.99.

- $\mathbb{P}_{>0.999}(\text{true}\,\mathbb{U}^{\leq 10}\,delivered)$, and also evaluates to true as the path formula evaluates to 0.9999.

- $\mathbb{P}_{>=1}(\text{true}\,\mathbb{U}\,delivered)$, evaluates to true.

The previous results are consistent with those obtained using the PRISM tool.

4.2.2 Craps Game

For the craps game, we can evaluate the following path formula:

- $Pr(start \models C\ \mathrm{U}^{\leq 2} B)$, where $C = \{start, 4, 5, 6\}$ and $B = \{won\}$. We evaluated this formula at 0,2608 ($\frac{338}{36^2}$) which is the same value evaluated in [2] (P. 768).

4.3 Performance Analysis

4.3.1 Birth-Death Process

We will now use a model of an M/M/1/N birth-death process [1] with $p_{N,N}$, λ_i and μ_i equal to $\frac{1}{2}$ for $0 \leq i \leq N$, where N is the model size, as illustrated in Figure 4.1. To evaluate the performance of our tool, we will consider 20 different values for N. Using our `dll` integration with Python (described in Section 3.7.2), we will create and evaluate 20 different model files corresponding to these 20 network sizes. Additionally, we will generate 20 formula files, each containing three PCTL formulas:

- Φ_1: `s1 : ( Pin]0,1/modelSize] ( mid ) U ( final ) )`
- Φ_2: `s0 : ( Pin]0,1E-10[ ( true ) U<=(N) ( final ) )`
- Φ_3: `s0 : ( Pin[0.5,0.5] X (s1) )`

where `final` is the label on the state ***N***, `start` is the label on the state ***0*** and `si` is the label on all the states, where `i` is the state's index. We also replace ***(N)*** by the size of the model.

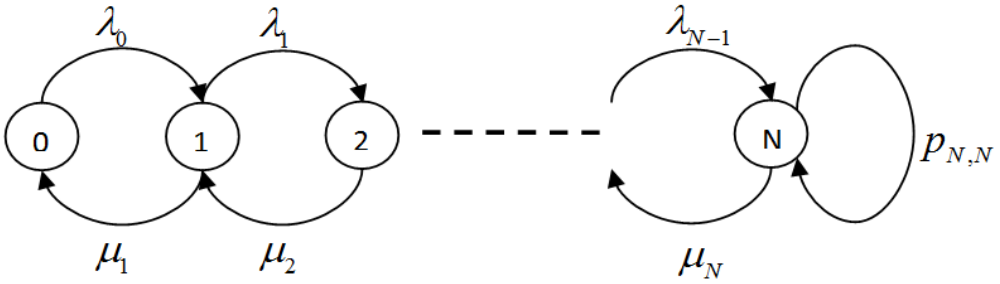

Figure 4.1: A Markov Chain Model for an M/M/1/N Birth-Death Process

N	**Time of Φ_1**	**Time ofΦ_2**	**Time of Φ_3**
100	0.013115	0.008809	0.000354
400	0.267055	0.424408	0.001315
700	1.452369	1.921505	0.003685
1000	3.771014	5.664472	0.006462
1300	8.383276	11.954765	0.009425
1600	16.718130	24.055948	0.016664
1900	28.498569	38.798166	0.029867
2200	42.044822	62.787263	0.028973
2500	66.395127	93.385719	0.047330
2800	87.325519	125.847799	0.064428
3100	123.773637	169.194850	0.072881
3400	157.105593	221.173322	0.088402
3700	220.584330	297.601152	0.115953
4000	242.006255	348.580019	0.121219
4300	310.851273	420.309127	0.149066
4600	373.741378	547.876274	0.197118
4900	485.681482	675.198405	0.215212
5200	560.878811	762.582375	0.235134
5500	655.308628	917.847000	0.254656
5800	732.259680	1027.687399	0.299438

Table 4.1: Execution Times in Seconds for the Different Models

As illustrated in Table 4.1, the execution time increases with the model size. Upon analyzing the execution times for the first and second formulas, we determined that our program exhibits a third-degree polynomial time complexity, as seen in Figure 4.2, for Φ_1 and Φ_2. This finding aligns with our expectations, since we employ a Gaussian elimination function for evaluating the `UnboundedUntilFormula` in Φ_1, which operates with a complexity of $O(n^3)$ in addition to the algorithm we use to find the set of states $S_{=1}$ which might have a complexity of $O(n^3)$ in the worst-case scenario as we are using three nested searching methods (`List.Any()` or `List.Where()`) to search within all the states of the model.

For the `BoundedUntilFormula` in Φ_2, we utilise a least fixed-point algorithm involving matrix multiplication, between a matrix and a vector, implemented with

a complexity of $O(n^2)$. Furthermore, due to the number of times, we execute this multiplication corresponding to the hop count, in our scenario with the 10 model sizes examined, the complexity increases to $O(n^2 \cdot m)$, where n is the model size and m is the hop-count. Taking into account that the hop count in Φ_2 is equal to the model size, we can see how the complexity is, in fact, $O(n^3)$, where n is the model size. Both the Gaussian elimination function and the multiplication in the LFP algorithm are primary contributors to the execution time of the `UnboundedUntilFormula` and the `BoundedUntilFormula` and represent bottlenecks in our program.

For the third formula Φ_3, it also involves matrix multiplication. However, due to one of the operands being a vector, the complexity is $O(n^2)$, resulting in comparatively shorter execution times as we see in the table.

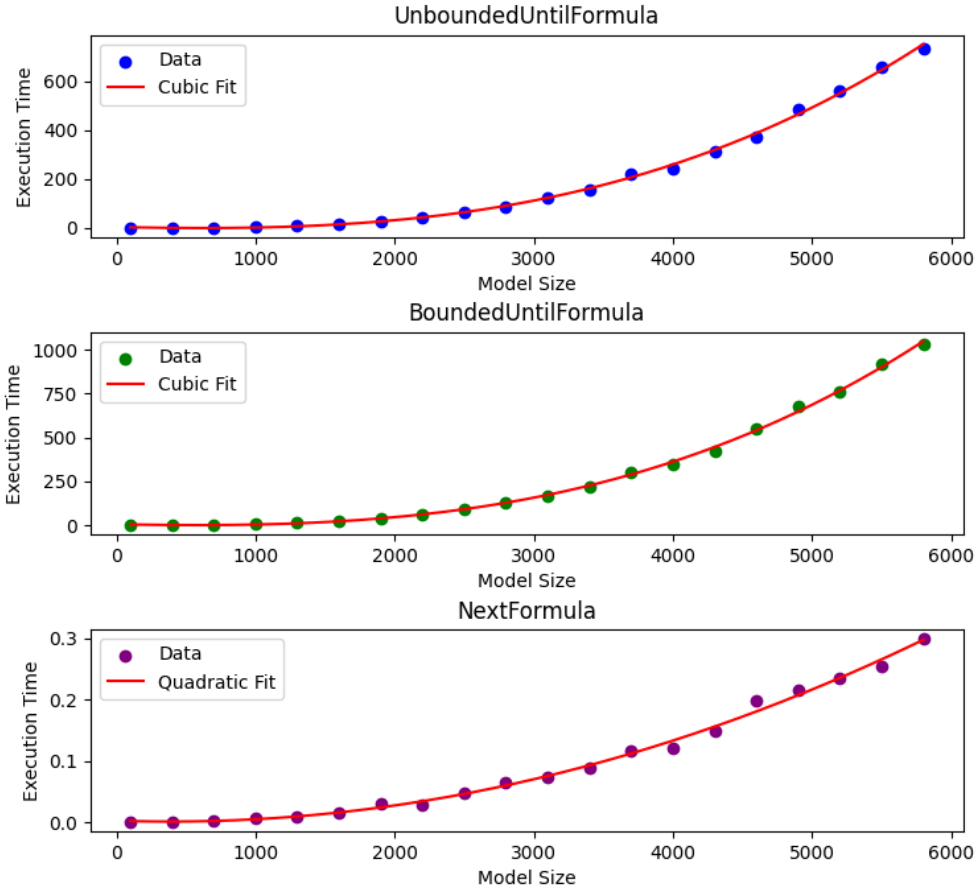

Figure 4.2: Visualization of the Execution Times for Evaluating Φ_1, Φ_2 and Φ_3

The figure above illustrates the execution times for Φ_1 (with `UnboundedUntilFormula`), Φ_2 (with `BoundedUntilFormula`), and Φ_3 (with `NextFormula`) corresponding to the data presented in the preceding table. Using `numpy.polynomial` in Python, we applied polynomial fits to the three datasets. Our analysis revealed that Φ_1 and Φ_2 fit a 3rd-degree polynomial, and Φ_3 fits a 2nd-degree polynomial. These findings align with the complexity of our code, which was discussed previously.

4.3.2 Worst Case Scenarios

As discussed in the previous section, we recognize that certain functions in the program exhibit high complexity based on their implementation. However, the previously reported numbers do not represent the worst-case scenario.

The program can take even longer to evaluate certain formulas. After testing various formulas on the earlier birth-death example, we observed that the function used to determine the $S_{=1}$ set, for the `UnboundedUntilFormula`, has a worst-case scenario when both `StateFormula`s, Φ_1 and Φ_2, are satisfied by all states in the model. Furthermore, the function used to find the vector $Sat(\Phi)$ also experiences a worst-case scenario when the `StateFormula` is satisfied by all states in the model.

In the future, we can focus on optimising these search algorithms to reduce their execution time, thereby simplifying the evaluation process for larger models.

Next, we will move on to the conclusion chapter to summarize all the work done in this thesis.

5 Conclusion

In this thesis, we discussed different concepts on Markov chain models, PCTL formulas, and also how to evaluate them mathematically. We then introduced a tool we developed to read Markov chain models and formulas from `.txt` files and evaluate PCTL formulas for a given state on the models. This evaluation helps verify that the models satisfy the specified formulas and behave as expected.

We demonstrated that our tool is user-friendly, functions as intended, and correctly evaluates formulas. Additionally, we showcased the tool's features and performance, identifying its major bottlenecks. The primary challenges were the method used to solve the system of linear equations and the implementation of matrix multiplication.

For future development, we recommend integrating libraries that can solve equations and perform matrix multiplications more efficiently. Additionally, implementing a graphical user interface (GUI) would enhance the usability of our tool, allowing users to visualize results and interact with models and formulas more easily. These improvements will significantly enhance the tool's performance and user experience.

Acronyms

CLI Command Line Interface. 21

DLL dynamic link library. 39

DTMC Discrete-Time Markov Chain. 5

ICT Information and Communications Technology. 4

IDE Integrated Development Environment. 21

OOP object-oriented programming. 21

PCTL Probabilistic Computation Tree Logic. 10, 21, 43

PRISM Probilistic Symbolic Model Checker. 43

VB.NET Visual Basic .NET. 21

Bibliography

[1] Alotaibi, Fahad ; Ullah, Israr ; Ahmad, Shakeel: Modeling and Performance Evaluation of Multi-Class Queuing System with QoS and Priority Constraints. In: *Electronics* 10 (2021), 02, S. 1–26. http://dx.doi.org/10.3390/electronics10040500. – DOI 10.3390/electronics10040500

[2] Baier, Christel ; Katoen, Joost-Pieter: *Principles of Model Checking*. Bd. 26202649. MIT Press, 2008. – ISBN 978–0–262–02649–9

[3] Barros, Gustavo M.: *NCalc, Mathematical Expressions Evaluator for .NET*. https://ncalc.github.io/ncalc/

[4] developers, NetworkX: *NetworkX documentation*. https://networkx.org/documentation/latest/. Version: 2024

[5] docs, MDN web: *Regular expressions*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions

[6] Microsoft: *Console Class*. https://learn.microsoft.com/en-us/dotnet/api/system.console?view=net-8.0

[7] Microsoft: *Floating-point numeric types*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types

[8] Microsoft: *Visual Basic documentation*. https://learn.microsoft.com/en-us/dotnet/visual-basic/. Version: 2019

[9] Microsoft: *Windows documentation*. https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library. Version: 2019

[10] NASA: *Deep-Space1 Spacecraft*. https://www.jpl.nasa.gov/nmp/ds1/tech/spacecraft.html

[11] Parker, Dr. D.: *Lecture 5 : PCTL Model Checking for DTMCs*. https://www.prismmodelchecker.org/lectures/pmc/05-dtmc%20model%20checking.pdf

[12] Parker, Dr. D.: *PRSIM docs*. https://www.prismmodelchecker.org/doc/

[13] Python.NET: *Python.NET documentation*. https://pythonnet.github.io/pythonnet/

[14] Varyani, Yash: *Gaussian Elimination to Solve Linear Equations*. https://www.geeksforgeeks.org/gaussian-elimination/