Software Project M60: Simulator for the Probabilistic Model Checker PRISM

by

Andrew Hinton

MEng in Computer Science/Software Engineering
Supervisor: Prof. Marta Kwiatkowska
School of Computer Science
University of Birmingham
April 2005

Abstract

PRISM is a probabilistic model checker which allows the modelling and analysis of systems that exhibit probabilistic behaviour. Such tools have been developed to allow the analysis of specifications which require performance and reliability measures, as well as correctness.

The project described in this dissertation adds to the functionality of PRISM by providing a simulator - a tool which calculates and reasons about execution paths through probabilistic models. The first part of the project was to design and implement a simulator user interface, allowing users to manually or automatically generate and visualise execution paths through their models. This user interface has already been used extensively by members of the PRISM development team to debug their PRISM models and to help other users with their problems. Feedback from such users indicate that the PRISM simulator is very useful as a debugger of PRISM models and specifications, but also as a means of understanding the behaviour of models and even the behaviour of PRISM itself.

The second part of the project was to investigate and develop efficient algorithms which use statistical sampling techniques to perform approximate model checking of temporal logic specifications. The solution allows the user to specify the degree of approximation and the amount of confidence to which they require their results. One of the key motivations for these techniques is that they do not need to enumerate the entire state space, and thus can handle larger models. The success of these techniques is demonstrated by their application to models that are much larger than what PRISM can currently handle on a standard PC and by a comparison of their results to what can be calculated by PRISM.

Keywords: Probabilistic model checking, Monte Carlo simulation, approximate probabilistic model checking, formal verification, discrete event modelling.

Acknowledgements

My deepest thanks go to Dr. David Parker, who not only gave up countless hours to the supervision of this project, but also acted as a system tester, domain expert and source of enthusiasm and support. I would also like to thank Prof. Marta Kwiatkowska for her guidance, the many users of the system who provided invaluable feedback, Hazel, Graham, Mike and Vipul for proof reading my dissertation and Helen for helping me with the biology case study. Last but not least I would like to thank my friends and family for their patience and support throughout this project and my years at university.

Contents

Chapter 1

Introduction

Formal verification is a branch of computer science concerned with checking whether computerised systems meet their specifications. One approach towards formal verification involves building a model to represent a system, and then performing a systematic analysis on that model to check whether it meets its specification. This approach is known as *model checking*, and it is of particular interest, because it can be automated.

Probabilistic model checking is aimed at analysing systems that exhibit probabilistic behaviour. It was derived from the need for a greater range of model and specification formalisms to accurately represent real-life systems. Specifications can therefore express performance and reliability measures, as well as correctness.

1.1 The PRISM Tool

PRISM [3] is a probabilistic model checker being developed at The University of Birmingham. It allows the modelling and analysis of systems that exhibit probabilistic behaviour. The tool itself accepts two files as input. The first is a model file; written in the PRISM language, this file contains a description of the system to be analysed. The second is a specification file; written in an appropriate temporal logic, it contains a list of properties to be checked against the system described in the model file.

The tool itself [5] is encapsulated within a graphical user interface that allows users to develop and build model files and also specify the property files to be checked. It also supports the idea of *experiments*; this is where results are calculated for ranges of input parameters (e.g. time, queue size) and plotted as a chart.

To illustrate the PRISM tool in action, Figure 1.1 shows an experiment on a model of a biological system. In this case [7], the cell cycle of eukaryotes is modelled and the probability of the amount of cyclin, an important protein in the cell cycle, existing in a particular state is investigated over time[1].

As well as biological modelling, PRISM has been successfully applied to a wide range of case studies [3] including:

- The quality of service of multimedia and communications protocols such as Bluetooth device discovery and IEEE 1394 FireWire Root Contention;
- The correctness and performance of probabilistic security protocols such as the Crowds protocol;
- The reliability of NAND multiplexing.

1.2 The PRISM Simulator

This project adds to the functionality of PRISM by providing a simulator to allow further analysis of probabilistic models. Simulation involves the generation of *paths* through a model. These paths can either be generated manually or automatically.

[1] Cyclins have the ability to bind themselves to other proteins and the state of the resulting complex determines the path of the cell cycle.

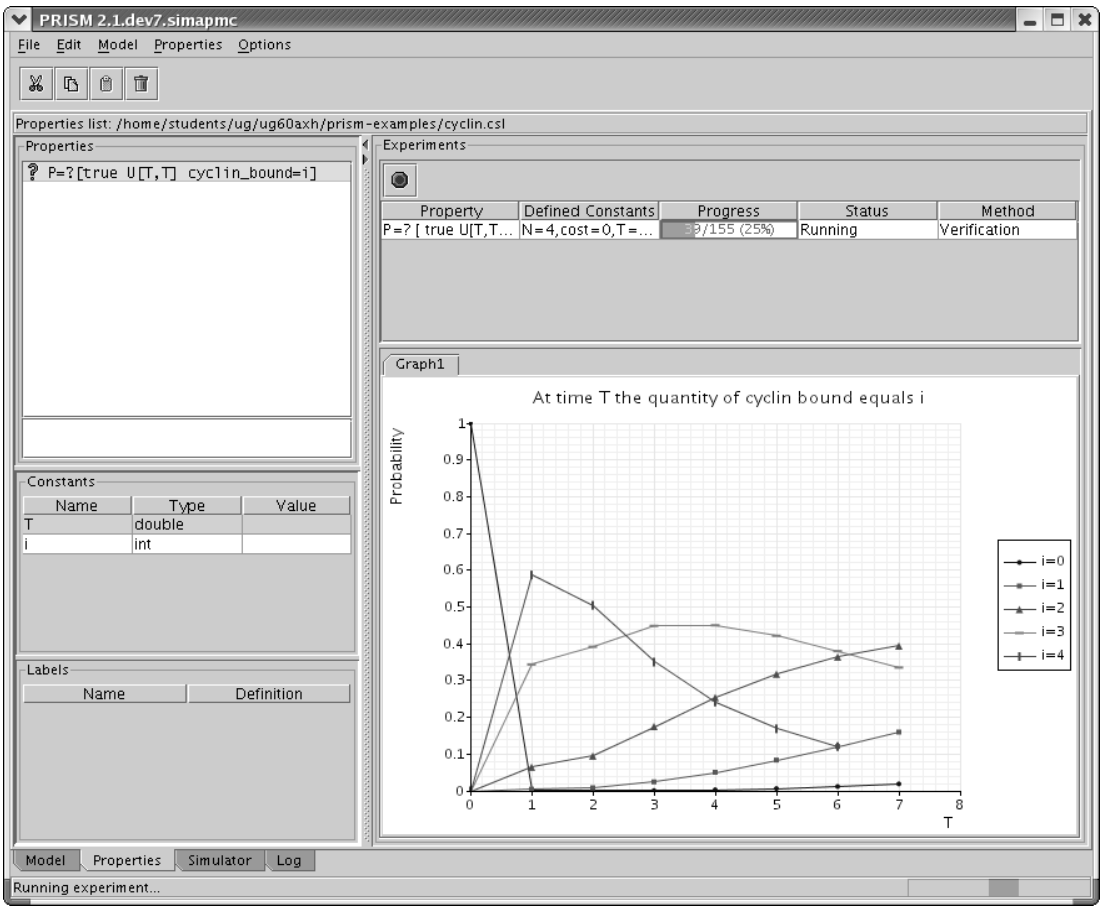


Figure 1.1: An experiment in the PRISM graphical user interface.

To illustrate this, Figure 1.2 shows a probabilistic automata of a fair dice, modelled using fair coin flips. When the automata undergoes a transition into state 7, d is assigned a value from 1 to 6.

Some example paths through this model are given below:

$$0 \to 2 \to 5 \xrightarrow{d'=4} 7$$

$$0 \to 2 \to 6 \to 2 \to 5 \xrightarrow{d'=4} 7$$

$$0 \to 2 \to 6 \to 2 \to 6 \to 2 \to 5 \xrightarrow{d'=4} 7$$

$$\vdots$$

$$0 \to 2 \to 6 \to 2 \to 6 \to \ldots \to 2 \to 5 \xrightarrow{d'=4} 7$$

A simulator provides the means of generating these *sample* paths by either:

- Allowing the user to pick updates to the current state. For example, in state 0, the user should be able to select updates from the set $\{state' = 1, state' = 2\}$;
- Automatically selecting updates to the current state. For example, in state 0, the simulator should produce a uniform random sample from the probability distribution $\{0.5 : state' = 1, 0.5 : state' = 2\}$ and apply the corresponding update.

The ability to produce sample paths enables the user to manually explore the state space of their model. This has the following benefits:

- It provides a good sanity check for users as they develop and debug their models.

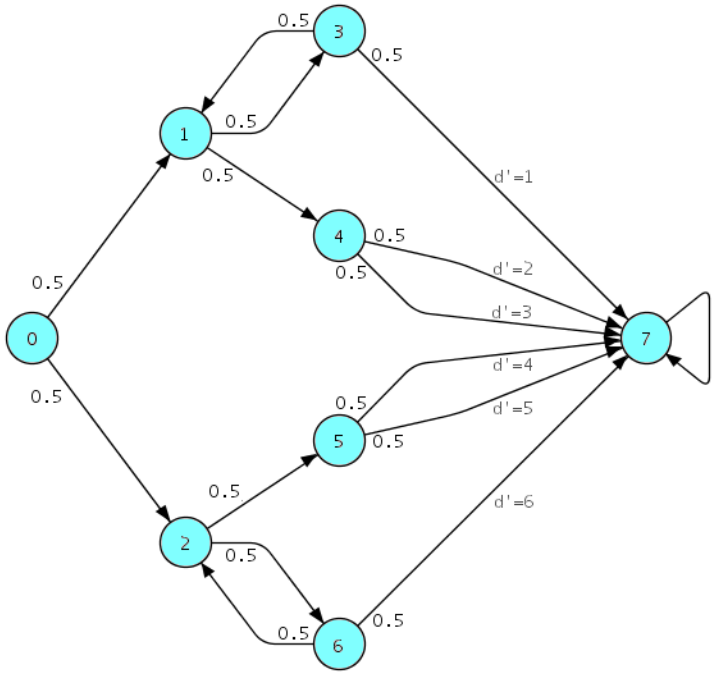


Figure 1.2: A probabilistic automata of a coin-flipping model of a dice.

- Users may be able to spot errors and/or understand the behaviour of their model more easily if it is displayed visually.

For each path, it is also possible to determine whether certain *properties* have been satisfied, allowing the user to investigate temporal logic specifications. In the dice model example, it may be desirable to determine the probability of throwing a 4 (i.e. when $state = 7, d = 4$). It is easy for a simulator to determine whether a particular path satisfies this. All that is required is the simple check: $(state = 7) \rightarrow (d = 4)$. However, determining the actual probability is not so simple.

The probability of a path through a probabilistic model is the product of the probabilities associated with every transition along that path. Therefore, the probability of throwing a 4 is the sum of the probabilities of all paths that end up in the state $state = 7, d = 4$.

However, as illustrated above, there are infinitely many paths to state 7 that assign $d' = 4$, making the probability:

$$\begin{aligned} \mathcal{P}(\text{throw 4}) &= \lim_{n\to\infty} \sum_{k=0}^{n} 0.5 \times (0.5^{2k}) \times 0.5 \times 0.5 \\ &= 0.125 + 0.125 \times 0.5^2 + 0.125 \times 0.5^4 + \ldots \\ &= 0.1\dot{6} \end{aligned}$$

PRISM solves this problem by reducing it to that of a linear equation system. While PRISM's implementation [8] is efficient and accurate, a consequence of building certain, more complex models into memory is that build times and memory usage can be great. This sometimes makes this type of numerical analysis impractical.

Another benefit of simulation is that it can be combined with statistical techniques [9], to be used as an alternative to numerical analysis. With the dice model example, suppose N sample paths are generated by the simulator and that M paths satisfy $state = 7 \rightarrow d = 4$. As $N \rightarrow \infty$, the probability of throwing a 4 is:

$$\mathcal{P}(\text{throw 4}) = \frac{M}{N}$$

The key advantage of this technique is that it only requires a single state of the model at a time, saving memory. Although this technique will yield less accurate results than numerical analysis, by controlling N, it is possible to obtain results within predefined error bounds.

A further type of analysis that is possible in PRISM is that of determining whether a particular path has accumulated any reward or cost. For example, suppose a reward of 1.0 is accumulated for each coin toss in the dice model example. PRISM is able to calculate measures such as the expected reward to reach the terminating state (i.e. when $s = 7$).

A simulator can approach this problem in a similar way to calculating the probability of reaching a certain state. With the dice model example, suppose N sample paths are generated by the simulator and that each sample path S_i, accumulates a reward R_{S_i} whilst reaching the terminating state, as $N \to \infty$, the expected reward will be:

$$\mathcal{R}(\text{terminate}) \quad = \quad \frac{\sum_{k=0}^{N} R_{S_k}}{N}$$

1.3 Aims and Objectives

The aim of this project was to design and implement a simulator for PRISM. The following objectives were outlined:

- Investigate, design and implement efficient data structures and algorithms for:
  - The automatic and manual generation of paths through each type of PRISM model;
  - The verification and analysis of properties over generated paths;
  - The calculation of rewards associated with generated paths as well as specific states of the model;
  - The computation of probabilities or expected rewards of certain properties, based on statistical sampling techniques.
- Design and implement an intuitive, but powerful graphical user interface for:
  - The automatic and manual exploration of PRISM models;
  - Allowing property verification using statistical sampling techniques, with the results also being made available for use in the PRISM experiments user interface.
- Investigate the use of error bounds [6, 9] for use with the statistical sampling techniques and implement if feasible.

1.4 Personal Motivations

I have been involved with the PRISM project since the summer of 2002; I helped to develop the current graphical user interface, graphical extensions to the PRISM language and a prototype simulator. I felt that undertaking this project would enable me to apply my acquired knowledge of probabilistic model checking and of the PRISM tool to a challenging project. Given my recent experiences with the model development process, one of my main motivations was that I would be producing a tool that would be of real benefit to PRISM users. Also, a key motivation was the opportunity to further develop the prototype simulator - a piece of work that required a considerable amount of time, effort and research.

1.5 Layout of the dissertation

This dissertation is structured as follows. Chapter 2 reviews tools and research that are related to simulation with PRISM, outlining where this project contributes to the area. Chapter 3 gives all background material for this dissertation and an analysis of that material for the purposes of eliciting requirements. These requirements are given as a specification in Chapter 4 before a detailed account of the design and implementation of the PRISM simulator is given in Chapter 5. Several techniques were used to ensure that the tool was thoroughly tested at each stage of

the software life-cycle - these are discussed in Chapter 6. The details of how the development of the PRISM simulator was managed throughout the software life-cycle is given in Chapter 7. Chapter 8 gives an evaluation of the various components of the PRISM simulator, before finally, Chapter 9 gives a summary of the achievements and deficiencies of the project.

Chapter 2

Review of Related Work

This chapter summarises previous work involving simulation and PRISM. Sections 2.1 and 2.2 investigate previous implementations and prototypes of simulators written specifically for PRISM. Their main features are evaluated along with their strengths and weaknesses. Section 2.3 investigates a tool called the Approximate Probabilistic Model Checker (APMC). This tool provides support for approximate verification of discrete-time Markov chains, one of the three types of PRISM model. Finally, Section 2.4 investigates research performed on the approximate verification of continuous-time Markov chains, another of the model types supported by PRISM.

2.1 PrismSim

PrismSim [4] was developed as a final year BSc project at the University of Birmingham. It provides support for manual and automatic exploration of discrete-time Markov chains and Markov decision processes within a graphical user interface. Developed in Java, it uses PRISM's underlying data structures of model files to:

- Display simulation execution paths;
- Allow users to select from a set of updates to the current state;
- Allow users to request that an update be made automatically.

PrismSim can be used to generate a set of sample paths of a fixed length. These paths are sent to a set of files for analysis via a regular expression checking engine. This engine provides facilities to model check a subset of a temporal logic known as probabilistic computation tree logic (PCTL) and to perform an analysis of the coverage of the state space.

The main disadvantage of PrismSim is its approach to model checking. The regular expression checking technique proved slow and inefficient[4]. The reason for this was that path generation was separated from path analysis. Therefore, if only a small proportion of a sample path is required to perform the analysis (for example, if the property being checked has already been satisfied, or if the path has entered a loop), a lot of computation time is wasted. Also, PrismSim was only applied to a small number of case studies and doesn't provide evidence that it scales up to larger examples.

Furthermore, this tool does not provide support for continuous-time Markov chains, the third type of model supported by PRISM, and is currently out of date with regards to current PRISM releases.

2.2 Prototype PRISM Simulator

In order to design a suitable user interface for the PRISM simulator, a prototype was developed and integrated into the existing graphical user interface. The PRISM simulator prototype had the following objectives:

- Provide a simple, but powerful user interface design for manual and automatic generation of sample paths that is integrated into the existing PRISM graphical user interface;

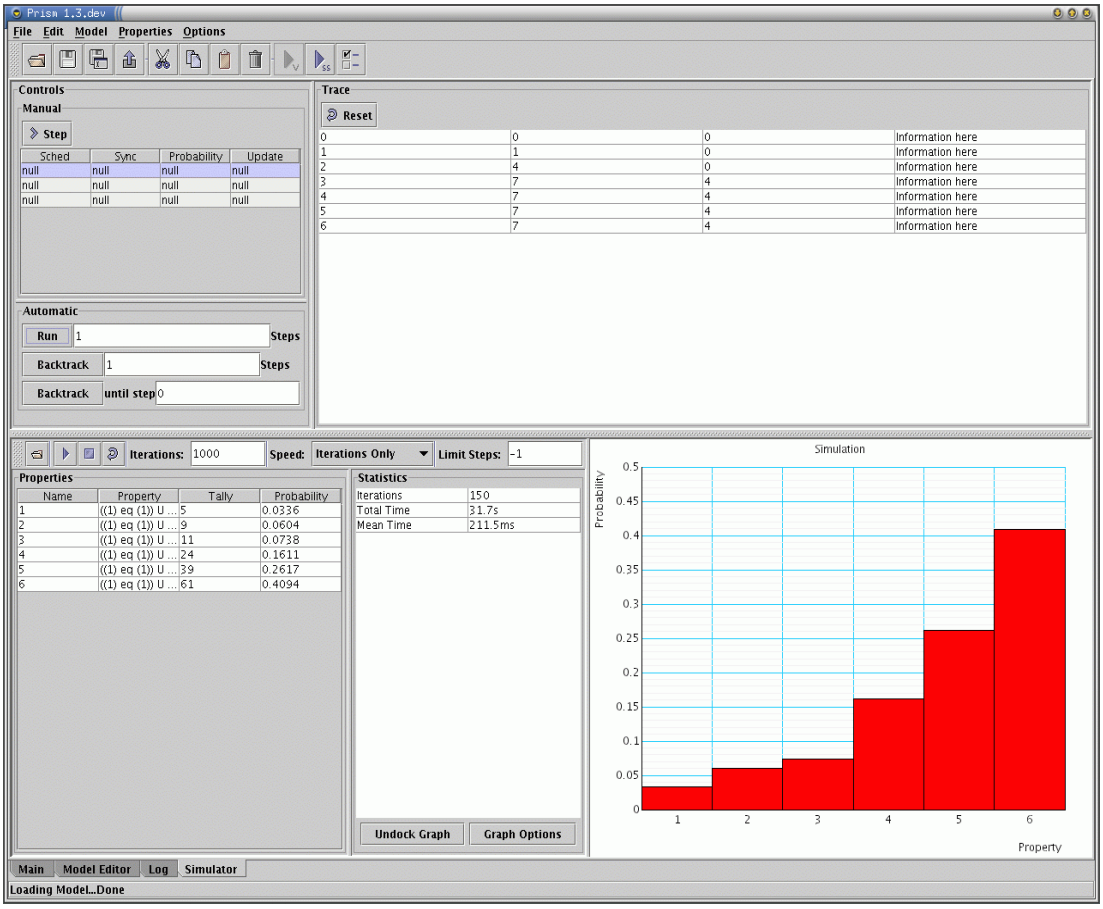


Figure 2.1: An experiment in the PRISM graphical user interface.

- Provide a program level interface to allow developers to provide simulator implementations using the user interface design.

In order to test the success of the program level interface, a DTMC simulator was developed. This provided support for automatic path generation. However, it did not provide support for important language features such as synchronisation and rewards and didn't allow manual path generation. The prototype simulator also provided the means to perform approximate model checking via a simple sampling technique. The results of the approximate probabilities could be viewed as a bar chart if appropriate. A screenshot of the prototype PRISM simulator is shown in Figure 2.1.

Despite the fact that the prototype PRISM simulator did not provide full support for the PRISM language and that it did not allow path generation with CTMC or MDP models, it did allow some of the key user requirements to be elicited and a number of user interface issues to be resolved. The sampling technique proved successful, but it could not be tested against more realistic models because of the lack of support for important language features.

2.3 The Approximate Probabilistic Model Checker

The Approximate Probabilistic Model Checker (APMC) [1] is a tool being developed at LRI, University Paris XI. APMC implements a randomised sampling algorithm known as the Generic Approximation Algorithm ($\mathcal{GAA}$) that "allows the efficient approximation of the satisfaction probability of monotone properties on probabilistic systems" [6]. An interesting feature of this algorithm is that the number of samples can be precalculated according to the required level of approximation and confidence.

The techniques used by APMC have recently been incorporated into PRISM. In particular:

- Discrete-time Markov chain models specified in the PRISM language can be converted into APMC data structures;
- Approximation verification of PCTL formulae can be performed within the PRISM graphical user interface[1].

2.4 Acceptance Sampling Research

Younes and Simmons [10] performed research into probabilistic verification of discrete event systems using a technique known as acceptance sampling. In order to perform a comparison of this technique with the numerical technique provided by PRISM, an implementation was provided that worked continuous-time Markov chain PRISM models.

A combined approach [9] was also implemented that allowed the acceptance sampling technique to be used alongside PRISM's model checking engines. This approach allowed nested formulae in a temporal logic known as continuous stochastic logic (CSL) to be checked. However, one disadvantage is that the benefits of not having to build the entire state space into memory are lost.

Despite the success of this research, it was never fully integrated into the PRISM tool and user interface.

2.5 Summary

This chapter has outlined simulation work and research related to PRISM. The following observations should be noted:

- None of the work provides a user interface that allows exploration of the state space of all three types of PRISM model;
- None of the work provides full support for all of PRISM's language features such as synchronisation and rewards;
- The work concerning the approximate verification of CTMCs is not integrated into the PRISM tool or user interface;
- The work concerning the approximate verification of DTMCs does not work with rewards based properties;
- None of the work is fully integrated with the latest graphical user interface, introduced in PRISM 2.0.

These points serve the purpose of highlighting the need for this project, as well as providing some of its most fundamental requirements. Therefore, it was the aim of this project to see that each of these points were addressed.

[1]The integration of APMC into the PRISM user interface was performed in parallel with this project. This allowed a common user interface to be provided for both.

Chapter 3

Background and Analysis

This chapter introduces all of the research and analysis that was performed in order to approach the problem of developing a simulator for PRISM. As well as summarising the background of the project domain, this chapter presents the necessary analysis of this material for application to the project. Much of the background material was gathered during the requirements elicitation process.

In Section 3.1, the probabilistic model checker PRISM is introduced in terms of the tool itself, and the different types of model it can analyse. Simulation involves the generation of paths through system models, and so Section 3.3 outlines how this can be done for each type of PRISM model. Section 3.4 describes how temporal logic specifications can be expressed in PRISM and how they should be verified over PRISM models. Finally, Section 3.5 outlines how a simulator can be used to perform approximate verification of these specifications.

3.1 The Probabilistic Model Checker PRISM

PRISM [3] is a probabilistic model checker, which allows the modelling and analysis of systems that exhibit probabilistic behaviour. The tool itself accepts two files as input. The first is a model file; written in the PRISM language, this file contains a description of the system to be analysed. The second is a specification file; written in an appropriate temporal logic, it contains a list of properties to be checked against the system described in the model file.

Systems can be modelled in PRISM as either:

- Discrete-time Markov chains (DTMCs);
- Markov decision processes (MDPs);
- Continuous-time Markov chains (CTMCs).

Properties can be specified using the following temporal logics:

- Probabilistic computation tree logic (PCTL) for DTMCs and MDPs;
- Continuous stochastic logic (CSL) for CTMCs.

PRISM builds a representation of the model into memory and performs numerical analysis to verify properties against it. This analysis is done by PRISM's various model checking engines. The output of the system is a set of results which state either:

- whether a property has been satisfied[1] -or-
- the probability of that property being satisfied.

This is summarised in Figure 3.1.

[1] PRISM in fact returns whether the property is satisfied in all reachable states

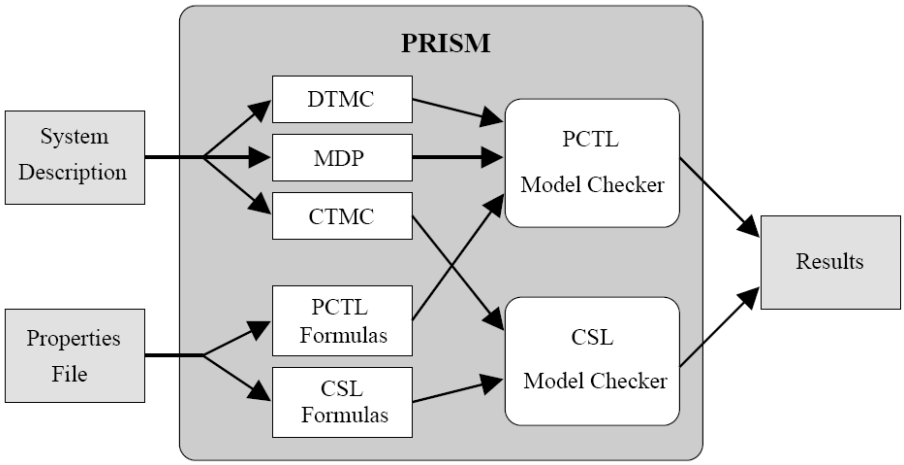


Figure 3.1: The structure of PRISM[3].

3.2 Probabilistic Models in PRISM

The three types of PRISM model can easily consist of several million states. It is not desirable to specify each of these states independently and so PRISM provides a high level system description language to specify these models more succinctly. Much of the PRISM language, including its syntax, is common to each type of PRISM model. Therefore, this section begins by describing the common attributes of the language before specific details for each type of PRISM model are given.

3.2.1 The PRISM Language

The PRISM language is a 'simple, module based description language' [2] which is used to create *system descriptions* for input into the PRISM tool. A system description is defined as 'the parallel composition of several interacting modules'[8] which can be expressed as a tuple $(\mathbf{M}, \mathbf{G}, \mathbf{R})$ where:

- $\mathbf{M}$ is a set of N modules, where each module $m \in \mathbf{M}$ can be thought of as a process that runs concurrently with other modules in the model. Each module contains a non-empty set, V_m of integer or Boolean variables;
- $\mathbf{G}$ is a set of global integer or Boolean variables;
- $\mathbf{R}$ is known as the *rewards construct*.

The current configuration of V_m is known as m's *local state* and the state of all variable sets, including the global variable set, $\mathbf{G} \cup V_{m_1} \cup V_{m_i} \cup \cdots \cup V_{m_N}$, is known as the *global state* or *current state*. The set, S, of all reachable global states is known as the *state space* of the model. It should also be noted that each module and variable is given a unique[2] name.

To describe transitions of the global state of the model, the PRISM language allows each module to contain a set, $Comm_m$ of *commands*. Each command, $c \in Comm_m$ for a particular predicate on the global state (known as the *guard*, $Guard_c$), describes how the local state and global variable set can be updated.

For a particular global transition, one or more modules can be scheduled to make a transition. When a module makes a transition independently, this is known as an *asynchronous* transition whereas if two or more modules make their transitions at the same time, this is known as a *synchronous* transition. Synchronous transitions can be specified by labelling two or more commands, from different modules, with an *action label* $a \in Act$ where Act is the set of all actions used in the model. Commands that are not labelled with action labels are specified as being *independent*. Intuitively,

[2]to all other module names, local variables, global variables, constants and action labels in the system description

> The set of all actions used by a module m_i, is denoted A_i. In each global state of the composed model, either one of the n modules makes an independent transition or, for some $a \in Act$, the set of modules $\{m_i \mid a \in A_i\}$ all make a synchronous, a-labelled transition.[8]

The rewards construct, $\mathbf{R}$, describes how rewards should be accumulated for particular states of the model, and for particular transitions between states. It can be defined as a tuple $(\mathcal{SR}, \mathcal{TR})$ where:

- $\mathcal{SR}$ is a set of *state reward items*. Each state reward item, $\rho \in \mathcal{SR}$, is associated with a predicate, $Guard_\rho$, and an expression, $Value_\rho$, which evaluates to a real number. When the model is in a global state that satisfies the predicate $Guard_\rho$, $Value_\rho$ is evaluated and accumulated as the reward value for that state.

- $\mathcal{TR}$ is a set of *transition reward items*. Each transition reward item, $\tau \in \mathcal{TR}$, is associated with a predicate $Guard_\tau$, an expression, $Value_\tau$, which evaluates to a real number and can be associated with an action label, $a_\tau \in Act$. Transition reward items accumulate rewards when a global transition is taken, and the predicate, $Guard_\tau$ evaluates to true. If the transition is asynchronous, the reward item must have no action label in order to accumulate the reward. If the transition is synchronous, a_τ must be equal to the transition's action label a_c in order to accumulate the reward. The value of the reward will be the result of the evaluation of $Value_\tau$.

Commands are interpreted differently for each type of PRISM model. The following three sections consider each model type in turn, describing how commands can be used to describe transitions between global states.

3.2.2 Discrete-Time Markov Chains

Discrete-time Markov chains (DTMCs) describe models that only consider the probabilities of transitions between global states. Figure 3.2 shows a simple DTMC. Each circle represents a different global state, with each line representing a transition between those states. Furthermore, each transition has an associated probability where the sum of the probabilities of the outgoing transitions from each state must be equal to 1.

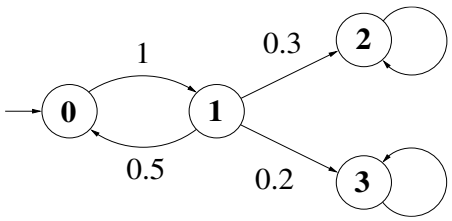
An example command is shown on line 9. In general, each command, c, describes the transitions that can occur when the model is in a global state that evaluates the predicate $Guard_c$ to true. In the example, the command describes the transition that should occur in all cases when variable s has the value of 1. After the $\rightarrow$ is a *probability distribution* of variable updates, $Updates_c$. Each update, $u \in Updates_c$ is separated by a '+' symbol and has the following structure:

- An expression, $\mathbf{P}_u$, that evaluates to the probability of that update, followed by a ':' symbol;

- A non-empty set of assignments, $Assign_c$, to variables separated by '&' symbols. Each assignment $as \in Assign_c$, specifies a primed variable and an expression describing the updated value of that variable.

For a DTMC, (ignoring synchronisation) only one command's guard is allowed to evaluate to true per module for each global state. This does not mean that a command cannot cover more than one global state. This is illustrated in the command on line 10 of Figure 3.2 where the command covers all global states where (s=2) or (s=3). If a state in S exists that is not covered by a guard of any command, it has no outgoing transitions and therefore is known as a *deadlock* state.

If there are N modules present in the model, and number of action labels present in the model is M it is possible that $1 \leqslant m \leqslant N + M$ command's guards will evaluate to true for a particular global state. In this case, the probability of each update is normalised by dividing it by m.




```
1.
2.  dtmc //model type
3.
4.  module figure3_2
5.
6.      s : [0..3] init 0; //local state
7.
8.      [] (s = 0) → 1 : (s′ = 1);
9.      [] (s = 1) → 0.5 : (s′ = 0) + 0.3 : (s′ = 2) + 0.2 : (s′ = 3);
10.     [] (s = 2) | (s = 3) → (s′ = s);
11.
12. endmodule
13.
```

Figure 3.2: An example Discrete-Time Markov Chain and the corresponding PRISM model.

3.2.3 Markov Decision Processes

The second type of PRISM model is the Markov Decision Process (MDP). MDPs describe systems that include non-determinism as well as probabilities when considering transitions between states of the system. MDPs are similar to DTMCs, except the sum of the probabilities of the outgoing transitions from each state need not be equal to 1. Rather than there being one probability distribution of outgoing transitions for each state, an MDP allows a set of probability distributions. Before a transition to another state can be made, one probability distribution must be selected non-deterministically before an update is selected probabilistically from that distribution.

In terms of the PRISM language, probability distributions can be specified as commands. The constraint that each probability distribution must have updates whose probabilities add up to 1 is still imposed. However, unlike DTMCs, more than one command is allowed to evaluate to true for each configuration of variables.

Figure 3.3 shows the PRISM code for a simple MDP. Lines 9 and 10 specify that in states that satisfy the guard (s=1), a non-deterministic choice should be made between two probability distributions.

3.2.4 Continuous-Time Markov Chains

The final type of PRISM model is the continuous-time Markov chain (CTMC). Like MDPs, CTMCs are a generalisation of DTMCs: they consider the time delay incurred before a transition occurs, allowing real-time considerations to be taken into account. Rather than associating a probability with each transition, a *rate* is associated instead. This rate describes how fast this transition should occur, the average time being: 1/rate.

For a particular state, there will be n transitions that could be made; each transition will have a rate, $\mathbf{R}_i$. The transition selected is the one that is *enabled* first. This is known as a *race condition* and it can be shown that the probability, $\mathbf{P}_k$ of selecting the kth transition is:

$$\mathbf{P}_k = \frac{\mathbf{R}_k}{\sum_{i=1}^{n} \mathbf{R}_i} \tag{3.1}$$

Once the transition has been selected, it is possible to determine the time delay, t, by taking

```
1.
2.  mdp //model type
3.
4.  module figure3_3
5.
6.      s : [0..3] init 0; //local state
7.
8.      [] (s = 0) → 1 : (s' = 1);
9.      [] (s = 1) → 0.5 : (s' = 0) + 0.3 : (s' = 2) + 0.2 : (s' = 3);
10.     [] (s = 1) → 1 : (s' = 1);
11.     [] (s = 2) | (s = 3) → (s' = s);
12.
13. endmodule
14.
```

Figure 3.3: An example Markov Decision Process in the PRISM language.

a sample from a negative exponential distribution, using the transition's rate as a parameter [8]:

$$t = \frac{-\log x}{\sum_{i=1}^{n} \mathbf{R}_i} \quad \text{where } x \text{ is a uniform random sample from the set } [0,1] \tag{3.2}$$

In terms of the PRISM language, transition rates can be specified in the same way that probabilities are specified in DTMCs and MDPs. Also, with CTMCs, transitions from the same state need not be specified in the same command. For example, the command:

$[]\ (s = 1)\ \rightarrow\ 10.0 :\ (s' = 0)\ +\ 20.0 :\ (s' = 2)\ +\ 1000.0 :\ (s' = 3);$

is equivalent to:

$[]\ (s = 1)\ \rightarrow\ 10.0 :\ (s' = 0);$
$[]\ (s = 1)\ \rightarrow\ 20.0 :\ (s' = 2);$
$[]\ (s = 1)\ \rightarrow\ 1000.0 :\ (s' = 3);$

Here, the numbers 10.0, 20.0 and 1000.0 are the rates of each update.

3.3 Execution Path Generation

Each of the three types of PRISM model can be described as *discrete event systems*. Younes and Simmons [10] give a generic framework for reasoning about execution paths through such systems. This framework is given in 3.3.1. The analysis performed to apply this framework to the three types of PRISM model is given in Sections 3.3.2 (DTMCs), 3.3.3 (MDPs) and 3.3.4 (CTMCs).

3.3.1 Path Generation in Discrete Event Systems

A discrete event system, such as a DTMC, MDP or CTMC, can be in one state $s \in S$ at any particular time, where S is the set of all states. The system remains in state s until an *event* occurs, where it moves into another state $s' \in S$. This transition of state is instantaneous (i.e. it takes 0.0 time for real-time models). Younes and Simmons define:

> "An *execution path* σ of a discrete event system, as a sequence:
>
> $$\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots$$
>
> with $s_i \in S$ and $t_i > 0$ being the time spent in state s_i before an event triggered a transition to state s_{i+1}." [10]

For the purposes of the PRISM simulator, it is necessary to extend this definition to incorporate the idea of rewards. Therefore, an execution path σ of a discrete event system is defined as a sequence:

$$\sigma = (s_0, sr_0) \xrightarrow{(t_0, tr_0)} (s_1, sr_1) \xrightarrow{(t_1, tr_1)}, (s_2, sr_2) \xrightarrow{t_2, tr_2} \dots$$

with sr_i being the reward accumulated in $s_i \in S$ and tr_i being the reward accumulated in the transition away from state $s_i \in S$.

The set of all paths starting in state $\bar{s} \in S$ is denoted as $Path(\bar{s})$. In order to generate elements of $Path(\bar{s})$, each model must define, for each state, s, how to get to other states in S. This is done by generating a set of possible events, E_s. The nature of these events and their generation will differ according to the type of model, but each event, $e \in E$ will be a tuple that defines the transition state s'.

Events can either be selected *manually* or sampled *automatically*. For manual selection, it is the responsibility of the user to choose an event e from E_s. Automatic sampling of e from E_s is different for each type of model. The transition state s' associated with e is then added to the current path as s_{i+1}, and for real-time models, the time spent in s is added as t_i.

Rewards are accumulated only when events occur. For example, it is only when the transition away from state s_i occurs that sr_i and tr_i are calculated. This is because, for real-time models, the amount of reward accumulated in a particular state, s_i is dependent upon the amount of time spent in that t_i. Assuming that for non-real-time models, $t_i = 1$, the total reward along a path $\mathcal{R}_\sigma$, is calculated as:

$$\mathcal{R}_\sigma = \sum_{i=0}^{n-1} (sr_i t_i + tr_i)$$

3.3.2 DTMC Path Generation

For discrete-time Markov chains in PRISM, an *event* can be considered as a tuple: $(\mathbf{P}, M, A, s')$ where:

- $\mathbf{P}$ is the *probability* of the event;
- M is the module that the event belongs to;
- $A \in Act$ is the action label that is associated with the event;
- s' is the new state.

For any current state, s, the procedure for generating the event set, E_s is as follows:

1. For each module, $m_i \in \mathbf{M}$, define a set, asynch_{m_i}, of all commands, where c is an independent transition and whose guards evaluate to true on s, i.e.

   $$\text{asynch}_{m_i} = \{c \in Comm_{m_i} \mid \text{c is independent, } Guard_c \xrightarrow{evaluates} \text{true}\}$$

   For each command, $c \in \text{asynch}_{m_i}$ and for each update, $u \in Updates_c$, add an event, e, to E_s where:

   - $\mathbf{P}_e$ = result of evaluating $\mathbf{P}_u$ on s;
   - $M_e = m_i$;
   - A_e = no action;
   - s'_e is the state after applying all assignments in $Assign_u$ to s.

2. For each action label, $a \in Act$, define a set, synch_a, of all commands that can be synchronised over a and whose guards evaluate to true on s, i.e.

   $$\text{synch}_a = \bigcup_{i=1}^{N} \{c \in Comm_{m_i} \mid a = \text{action label of } c, Guard_c \xrightarrow{evaluates} true\}$$

   For each module, $m_i \in \mathbf{M}$, if $\exists c \in Comm_{m_i}$ where the action label of $c = a$, the statement: "$\exists c \in \text{synch}_a$ where the $c \in Comm_{m_i}$" must be true, or else the action, a is said to be *blocked* in s. If a is not blocked, define a command, c_{synch_a} that is the *product* of all commands in synch_a. The product, $c_x \times c_y$, of two commands, c_x and c_y, is formed thus:

- The action label of $c_x \times c_y$ is the same as the action labels of either c_x or c_y, since both should be the same.
- $Guard_{c_x \times c_y}$ is never used and so is ignored.
- $Updates_{c_x \times c_y}$ is formed as the product of $Updates_{c_x}$ and $Updates_{c_y}$. The product, $u_x \times u_y$, of two updates, u_x and u_y, is formed thus:
  - The probability, $\mathbf{P}_{u_x \times u_y}$, will be equal to the multiplication of the probabilities of the two updates:
$$\mathbf{P}_{u_x \times u_y} = \mathbf{P}_{u_x} \cdot \mathbf{P}_{u_y}$$
  - The assignment set, $Assign_{u_x \times u_y}$, will be equal to the union of $Assign_{u_x}$ and $Assign_{u_y}$:
$$Assign_{u_x \times u_y} = Assign_{u_y} \cup Assign_{u_x}$$

For each update, $u \in Updates_{c_{synch_a}}$, add an event, e, to E_s where:

- $\mathbf{P}_e$ = result of evaluating $\mathbf{P}_u$ on s;
- M_e = no module;
- $A_e = a$;
- s'_e is the state after applying all assignments in $Assign_u$ to s.

3. If $\mathcal{N}$ is defined as the number of modules in $\mathbf{M}$ that have at least one command whose guard evaluates to true on s, i.e.:
$$\mathcal{N} = |\{m \in \mathbf{M} | \left(\exists c \in Comm_m \mid Guard_c \xrightarrow{evaluates} \text{true}\right)\}|$$
and $\mathcal{M}$ is defined as the number of action labels in Act that are not blocked in s, for each event, $e_i \in E_s$, *normalise* the probability:
$$\mathbf{P}'_e = \frac{\mathbf{P}_e}{\mathcal{N} + \mathcal{M}}$$

Automatic sampling from E_s is straightforward once it has been generated. Because for DTMCs, the sum of all probabilities for each event, e_i in E_s should be equal to 1 , i.e.
$$\sum_{e \in E_s} \mathbf{P}_{e_i} = 1$$
the probability of selecting, e_i from E_s is simply $\mathbf{P}_{e_i}$.

Once an event, e, has been sampled, it is simple to calculate the state reward, sr_i for state s_i in σ. sr_i is equal to sum of the evaluation of all $Value_\rho$ on s for all $\rho \in \mathcal{SR}$ where $Guard_\rho \xrightarrow{evaluates}$ true on s, i.e.
$$sr_i = \sum_{\rho \in \mathcal{SR}_{\text{true}}} Value_\rho \text{ where } \mathcal{SR}_{\text{true}} = \{\rho \in \mathcal{SR} \mid \text{Guard}_\rho \xrightarrow{evaluates} \text{true}\}$$

The calculation of the transition reward, tr_i, is done by obtaining the sum of the evaluation of all $Value\tau$ on s for all $\tau \in \mathcal{TR}$ where $Guard_\rho \xrightarrow{evaluates}$ true on s and $a_\tau = a_e$, i.e.
$$tr_i = \sum_{\tau \in \mathcal{TR}_{\text{true}}} Value_\tau \text{ where } \mathcal{TR}_{\text{true}} = \{\tau \in \mathcal{TR} \mid \text{Guard}_\tau \xrightarrow{evaluates} \text{true}, a_\tau = a_e\}$$

3.3.3 MDP Path Generation

For Markov decision processes in PRISM, an *event* can be considered as a tuple: $(\mathcal{D}, \mathbf{P}, M, A, s')$ where the definitions of $\mathbf{P}$, M, A and s' are the same as for DTMCs and $\mathcal{D}$ is the *index* of the probability distribution. $\mathcal{D}$ is required, because for a transition from any state, $s \in S$, in an MDP, a non-deterministic choice has to be made from a set of probability distributions and $\mathcal{D}$ is used to index the probability distribution that each event belongs to.

For any current state, s, the procedure for generating the event set, E_s is as follows:

1. This step is equivalent to Step 1 for DTMCs, except a unique index, index_c, is also associated with each command in each asynch_{m_i} set. For each command, $c \in \text{asynch}_{m_i}$ and for each update, $u \in \mathit{Updates}_c$, add an event, e to E_s where:

   - $\mathcal{D}_e = \text{index}_c$
   - $\mathbf{P}_e$, M_e, A_e and s'_e are the same as for DTMCs.

2. This step is equivalent to Step 2 for DTMCs, except a unique index (also unique to indexes assigned in Step 1) is associated with each synchronised command, c_{synch_a} for each action $a \in \mathit{Act}$ that has not been blocked.

3. Unlike DTMCs, there is no need to *normalise* the probabilities of the events in E_s.

Automatic sampling from E_s is not so straightforward as in the DTMC case. The problem lies with the selection of probability distribution, which is a nondeterministic choice. According to [8], these nondeterministic choices 'are made by an *adversary* (also known as a 'scheduler' or 'policy'), which selects a choice based on the history of choices made so far.' Under any adversary, the behaviour of a MDP is probabilistic.

However, analysis of MDPs using a single adversary is of limited use [8]. In fact, PRISM's numerical engines investigate 'the *maximum* or *minimum* probability that some specified behaviour is observed *over all possible adversaries*'. Unfortunately, deriving adversaries that calculate the maximum and minimum probabilities for particular behaviour would involve an enumeration of the entire state space of the model and this is something that simulation techniques aim to avoid.

One possible approach is to consider each nondeterministic choice as purely probabilistic. In this case, if the number of distinct probability distributions, $\mathcal{N}$, in E_s, i.e.

$$\mathcal{N} = |\{\mathcal{D}_e \mid e \in E_s\}|$$

Then, an index should be sampled from a uniform probability distribution where, $0 \leqslant \text{index}_{sample} < \mathcal{N}$. index_{sample} can now be used to select the probability distribution. Create a set E'_s where only events exist that have $\mathcal{D} = \text{index}_{sample}$, i.e.

$$E'_s = \{e \in E_s \mid \mathcal{D}_e = \text{index}_{sample}\}$$

The sum of all probabilities for each event, e_i in E'_s should now be equal to 1 , i.e.

$$\sum_{e \in E'_s} \mathbf{P}_e = 1$$

the probability of selecting, e_i from E'_s is simply $\mathbf{P}_{e_i}$.

This approach will not give accurate results when analysing property specifications over MDPs. However, this approach does provide a way to automatically generate sensible paths through MDPs, giving users a way to quickly explore the state space.

The calculation of state and transition rewards for MDPs is the same as for DTMCs.

3.3.4 CTMC Path Generation

For continuous-time Markov chains in PRISM, an *event* can be considered as a tuple: $(\mathbf{R}, M, A, s')$ where the definitions of M, A and s' are the same as for DTMCs and $\mathbf{R}$ is defined as the *rate* of the event.

For any current state, s, the procedure for generating the event set, E_s is as follows:

1. This step is equivalent to Step 1 for DTMCs. However, because each CTMC update, u, is associated with a rate, $\mathbf{R}_u$ rather than a probability, the procedure should use:

   - $\mathbf{R}_u$ instead of $\mathbf{P}_u$ when referring to commands;
   - $\mathbf{R}_e$ instead of $\mathbf{P}_e$ when referring to events.

2. This step is equivalent to Step 2 for DTMCs (with the modifications explained in Step 1). It should be noted that the calculations used to calculate synchronised event probabilities can be used in the same way to calculate synchronised event rates.

3. Like MDPs, there is no need to *normalise* the rates of events in E_s.

Automatic sampling from E_s involves calculating a probability, P_e, of each event according to its rate, R_e and the rates of all other events in E_s. Equation 3.1 describes how to calculate this because each event realises a transition of the CTMC.

For CTMCs, every time an event occurs, i.e. when the model makes a transition from s to s', the time spent in s is required. Equation 3.2 can be used to calculate this time.

As stated previously in Section 3.3.1, for real-time models such as CTMCs, the reward accumulated in a state, s_i, depends upon the time, t_i, spent in that state. Therefore, for CTMCs, there is a differentiation made between:

- State reward of s_i - This is the same as sr_i calculated for DTMCs and MDPs.
- Reward accumulated in s_i - This is calculated as $sr_i \cdot t_i$.

The *reward accumulated in* s_i is usually only used to calculate the total path reward as defined in Section 3.3.1.

Transition rewards are calculated in the same way as for DTMCs.

3.4 Property Specification

Property specifications are used in PRISM to analyse probabilistic models. Examples of property specifications are: "The probability that the algorithm will terminate is equal to 1", "What is the probability that the amount of sodium molecules will be less than 10 at the time of 1 millisecond" or "On average, the cost to run the machine for 1 year will not exceed 100,000". Such *properties* can be specified in an appropriate *temporal logic*.

PRISM uses two different temporal logics to reason about it's three types of probabilistic model. Probabilistic computational tree logic (PCTL) is used to specify properties of discrete-time Markov chains and Markov decision processes, and Continuous stochastic logic (CSL) is used to specify properties of continuous-time Markov chains.

In PRISM however, PCTL and CSL look very similar, the main difference being that PCTL reasons about discrete time steps and CSL reasons about real time intervals. Because of this similarity, a description of PRISM's property language is given where PCTL and CSL are discussed together. Extensions to PCTL and CSL such as rewards based properties and the direct querying of path properties are also considered. It should be noted that the outline given below is based on Section 4 of the PRISM manual [2].

3.4.1 State propositions

When reasoning about the behaviour of PRISM models, it is important to be able to 'identify particular sets or classes of states of the model' [2]. For example, consider the property: "What is the probability that the amount of sodium molecules will be less than 10 at the time of 1 millisecond". In order to verify this formula, there needs to be a way to specify the proposition: 'the amount of sodium molecules is less than 10'.

PRISM provides a way of specifying these propositions as Boolean *expressions*. In the previous example, if the PRISM model in question contains a variable, `num_sodium`, the corresponding expression would be:

```
num_sodium < 10
```

PRISM expressions are discussed in detail in Section 3.11 of the PRISM manual [2]. In future, `<expr>` will be used to describe Boolean PRISM expressions.

3.4.2 Property Syntax

Properties in PRISM can be much more expressive than simple state propositions. Properties are 'evaluated with respect to a single state of the model'. For a state $s \in S$, a property will evaluate to either true (satisfied in s) or false (not satisfied in s).

The syntax of PRISM state properties can be defined recursively. For the base case, a PRISM property, `<prop>`, can be:

- `true`, which is satisfied in all states;
- `false`, which is not satisfied in any state;
- `<expr>`, which is any Boolean PRISM expression;

Furthermore, `<prop>`, can be one of the following temporal operators:

- `P<op><p> [ <pathprop> ]`, which is a *path formula*. Path formulae will be discussed shortly.
- `S<op><p> [ <prop> ]`, which is a *steady-state formula*. Steady-state formulae will be discussed shortly.

and the following rewards operator:

- `R<op><p> [ <rewardprop> ]`, which is a *reward formula*. Reward formulae will be discussed shortly.

In order to specify more complex sets of states, PRISM allows several logical connectives:

- `!<prop>`, which is satisfied if the `<prop>` is not satisfied;
- `<prop> & <prop>`, which is satisfied if both `<prop>` properties are satisfied;
- `<prop> | <prop>`, which is satisfied if either of the `<prop>` properties are satisfied;
- `<prop> => <prop>`, which is satisfied if the first `<prop>` implies the second `<prop>`.

3.4.3 Path Formulae

Path formulae are used to reason about the probabilities of paths through PRISM models and are specified using the operator: `P<op><p> [ <pathprop> ]`. Intuitively, the meaning of this operator is that it:

> "is true in a state s of a DTMC, MDP or CTMC if the probability that path property `<pathprop>` is satisfied by the paths from state s meets the bound `<op><p>`." [2]

where `<op>` can be either `<`, `>`, `<=` or `>=` and `p` is used to refer to the value of the probability bound, and therefore must be a real number in the range $[0, 1]$.

There are three different types of path property, i.e. `pathprop` can be:

- A "Next" property: `X <prop>`;
- An "Until" property: `<prop> U <prop>`;
- A "Bounded Until" property: `<prop> U<time> <prop>`.

For any execution path, σ from a state $s \in S$, a path property will either evaluate to true or false. The following three sections discuss the details of this evaluation for each type of path property.

The "Next" Operator

For an execution path, σ, starting in state s_0, the path property `X <prop>` is true if `<prop>` is satisfied in the *next* state, s_1.

The "Until" Operator

For an execution path, σ, starting in state s_0, the path property `<prop1> U <prop2>` is true if `<prop1>` is satisfied for all states in σ *before* `<prop2>` eventually is satisfied. i.e. there exists an integer, i, such that `<prop1>` is satisfied for all states s_j where $j < i$ and `<prop2>` is satisfied in state s_i.

The "Bounded Until" Operator

For an execution path, σ, starting in state s_0, the path property `<prop1> U<time> <prop2>` is true if `<prop1>` is satisfied for all states in σ *before* `<prop2>` is satisfied in the bound specified by `<time>`.

The `<time>` is interpreted differently for different types of PRISM model. For DTMCs and MDPs, `<time>` must be of the form: `<= <t>` where `<t>` is an expression that evaluates to a non-negative integer. The bound is interpreted as that `<prop2>` must be satisfied in `<t>` discrete steps.

For CTMCs, `<time>` can be of the forms:

- `<= <t>`;
- `>= <t>`;
- `[<t1>, <t2>]`,

where `<t>`, `<t1>` and `<t2>` are non-negative real numbers and `<t1>`$\leqslant$`<t2>`. Therefore, there must exist an integer, i, such that `<prop1>` is satisfied for all states s_j where $j < i$ and `<prop2>` is satisfied in state s_i and $\sum_{k=0}^{i-1} t_k$ must be within the bound specified in `<time>`.

3.4.4 Steady-State Formulae

Steady-state formulae are used to reason about the long-run behaviour of CTMC models:

> "Informally the property: `s<op><r> [ <prop> ]` is true in a state s of a CTMC if starting from s, the steady-state (long-run) probability of being in a state which satisfies `<prop>`, meets the bound `<op><p>`." [2]

3.4.5 Rewards Formulae

Rewards formulae are used to reason about the expected reward of paths through PRISM models and the expected reward at particular instants in time and are specified using the following operator: `R<op><p> [ <rewardprop> ]`. The meaning of this operator is that it:

> "is true in a state s of a DTMC, MDP or CTMC if the expected reward associated with `rewardprop` of the model when starting from the state s meets the bound `<op><r>`." [2]

where `<op>` can be either `<`, `>`, `<=` or `>=` and `r` is used to refer to the value of the bound, and therefore must be a non-negative real number.

There are three different types of reward property, i.e. `rewardprop` can be:

- A "Reachability" reward: `F <prop>`;
- A "Cumulative" reward: `C <t>`;
- An "Instantaneous" reward: `I <t>`;
- A "Steady-state" reward: `S`

The following four sections discuss the details of the evaluation of each type of reward property.

The "Reachability" operator

Reachability rewards of the form `F <prop>` allow properties that determine the expected reward to reach a particular state or set of states. The reward along a path is different for each type of PRISM model.

For DTMCs and MDPs, the reward along a path, σ, that first satisfies the state formula `<prop>` in state s_i is:

$$\sum_{j=0}^{i-1} sr_j + tr_j$$

For CTMCs, the reward accumulated in each state is multiplied by the time spent in that state, therefore the reward along a path, σ, that first satisfies the state formula `<prop>` in state s_i is:

$$\sum_{j=0}^{i-1} sr_j \cdot t_j + tr_j$$

For all PRISM models, if `<prop>` is never reached, the reward is infinite.

The "Cumulative" operator

Cumulative rewards of the form `C<= <t>` allow properties that determine the expected reward to reach a given time bound. The reward along a path is different for each type of PRISM model.

For DTMCs and MDPs, `<t>` is an expression that evaluates to a non-negative integer, i. The reward along a path σ is:

$$\sum_{j=0}^{i-1} sr_j + tr_j$$

For CTMCs, `<t>` is an expression that evaluates to a non-negative real number, t. To calculate the reward along a path, σ, there needs for be a state s_i where $\sum_{k=0}^{i-1} t_k \geqslant t$. The reward is therefore:

$$\sum_{j=0}^{i-1} sr_j \cdot t_j + tr_j$$

The "Instantaneous" operator

Instantaneous rewards of the form `I=<t>` allow properties that determine the expected state reward at the given time instant. The reward along a path is different for each type of PRISM model.

For DTMCs and MDPs, `<t>` is an expression that evaluates to a non-negative integer, i. The reward is simply the state reward in state s_i which is sr_i.

For CTMCs, `<t>` is an expression that evaluates to a non-negative real number, t. To calculate the reward along a path, σ, there needs to be an integer, i where $\sum_{k=0}^{i} t_k \geqslant t$. The reward is simply the state reward is state s_i which is sr_i.

It should be noted that there is no need to multiply sr_i with the time spent in the state, as was the case for the previous rewards operators.

The "Steady-State" operator

The *steady-state* operator determines the expected reward in the long-run.

3.4.6 Determining the Actual Value

If the outer operator of a PRISM property is a path, steady state or rewards formula, it is possible to query the actual probability/reward of the property, rather than comparing it to some bound:

- `P=? [ <pathprop> ]`, which returns the probability of the path formula `<pathprop>` starting in the default initial state of the model [3].
- `S=? [ <prop> ]`, which returns the steady-state probability of being in a state which satisfies `<prop>`, starting in the default initial state of the model.

[3] It should be noted that PRISM allows properties to be verified considering any arbitrary initial state, or set of initial states.

- `R=? [ <rewardprop> ]`, which returns the actual reward of reward property `<rewardprop>`.

This style of property is important, because it is the only style that can be handled by the PRISM simulator.

3.4.7 Properties Files

Properties files are used by the PRISM tool to store lists of properties and various other information:

- Constants - These can be of type `int`, `double` or `bool` and can be used as part of expressions in properties. Constants can be left undefined so that they can be assigned a specific value or range of values for model checking.
- Labels provide a mechanism for defining and reusing propositions. For example, if many properties refer to a *terminating* state, a label can be used to describe this:

  ```
  label "terminating" = state=0 & s>10
  ```

  meaning that the property, "the algorithm eventually terminates with probability 1" can be expressed as:

  ```
  P>=1 [ true U "terminating" ]
  ```

3.5 Approximate Probabilistic Model Checking

This section describes techniques based on Monte Carlo simulation and statistical sampling that can be used to perform approximate model checking of PRISM models. The material presented is an adaptation for PRISM of the work presented in [6], in particular the *generic approximation algorithm*, $\mathcal{GAA}$. Furthermore, this algorithm is extended to take into account the existence of loops in paths.

Section 3.5.1 describes the subset of PRISM properties that can be analysed by simulation techniques. Section 3.5.2 continues by outlining the generic approximation algorithm and Section 3.5.3 describes how this algorithm can be extended the deal with PRISM models.

3.5.1 Which Properties can/cannot be Simulated?

Simulation involves the generation of *paths* through models. Therefore, it is possible to say, for a particular path whether a path formula such as, `[ true U<=k d=0 ]` is true. All that the simulator would need to do is generate a path of length `k` by:

- Setting s_0 to some initial state $\bar{s}$;
- While $i <$ `k`, repeat the following (as described in section 3.3):
  - Generate the set E_{s_i};
  - Automatically sample an event e from E_{s_i},
  - Set s_{i+1} to be s'_e and calculate the state and transitions rewards, sr_i and tr_i.
  - For CTMCs, also calculate the time spent in s_i and set as t_i.

This path satisfies `[ true U<=k d=0 ]` if there exists a state where the expression `d=0` evaluates to true. Moving on from this, it is also possible to determine an approximation of the probability of this path formula, i.e. `P=?[ true U<=k d=0 ]`. If N paths are generated in the way described above, and M paths satisfy `[ true U<=k d=0 ]`, as $N \to \infty$, the probability will be M/N.

As well as "bounded until" formula, this approach is applicable to "next" formula and also to "cumulative" and "instantaneous" rewards formula. There is a slight difference for CTMCs where instead of generating paths of fixed length, paths are generated until the total path time, i.e. $\sum_{k=0}^{i-1} t_k$, exceeds the bound described in the formula.

Unbounded Formulae

The first problem arises with unbounded formulae such as "until" path properties and "reachability" reward properties. Suppose there is a model where the property: `P=?[ true U d=0 ]` evaluates to the probability 0.5. This means that if N infinitely long sample paths are generated, $0.5N$ of these will contain a state s_j that satisfies the proposition, $d = 0$. For paths that do satisfy this proposition in state s_j, it is only necessary for a simulator to generate that path for j steps. However, a simulator cannot generate infinite paths and so the issue arises as to where to stop looking for the satisfaction of $d = 0$.

Suppose paths are generated until either the $d = 0$ is satisfied, or until the path reaches a predefined length, k where it is said that the formula is not satisfied. If for a particular state, s_j, $d = 0$ is satisfied, but $j > k$, the resulting probability will be incorrect.

Therefore, unbounded formulae can only be verified by a simulator if they are *monotone*. Monotone formulae are those which have the property where the 'truth of the formula at length k implies truth in the entire model' [6]. It is the responsibility of the user to ensure that k is large enough to ensure that their formula is monotone.

Steady-State Formulae

The second problem arises with steady state formulae such as `S=?[ d=0 ]` or `R=?[ S ]`. This is because their evaluation would require a state-based analysis, and the strength of simulation techniques are their ability to do path-based analyses. For this reason, steady-state formulae are not considered by the PRISM simulator.

Nested-Path Formulae

Consider properties that have a nested path formula such as:

- `P>=1[ true U P<0.5[ d=0 U<=5 g=10 ] ]`, which means, "the probability is 1 that the system will eventually be in a state where from that state the probability is less than 0.5 that a path will exist where d is equal to 0 until within 5 time units g is equal to 10";
- `R=?[ F P>=1[ true U d=0 ] ]`, which means, "what is the expected reward to reach a state where from that state all paths will eventually reach a state where d is equal to 0".

In order to verify such formulae over a particular path, σ, of length k, it would be necessary to verify the inner formula for each state in σ, $s_0 \dots s_k$. This would mean generating another N sample paths for each state in order to determine the truth of the probability bound. This increases the complexity of the algorithm from $\mathcal{O}(N \cdot k)$ to $\mathcal{O}N^2 \cdot k^2$, which, considering values of N are typically of the order 10^5 and values of k are typically of the order 1×10^3, is a significant, and can be considered an intractable, increase. Furthermore, the complexity increases by powers of N and k for each level of nesting.

One possible approach to this problem is to let the PRISM numerical engines calculate the set of states that satisfy the nested formulae. Then, for each state in σ, one would only have to check whether each state, s_i is in that set. This is the approach used in [9]. It is particularly advantageous if PRISM finds it computationally inexpensive to verify the inner formula, but requires very large amounts of time or memory to verify both inner and outer formulae together.

The disadvantage of this approach is that the model has to be built into memory, and so the advantages of simulation techniques, such as only requiring one state of the model at a time, are lost.

Initial State Considerations

When PRISM evaluates a property (excluding `P=?`, `S=?` and `R=?`), it returns true only if that property is satisfied in all states. Also, it is possible specify whether the property should only be satisfied in a subset of states. For example, PRISM includes a default label, `"init"`, which represents the set of initial states specified in the model, so it is possible to say:

```
"init" => P<=0.5 [ true U s=0 ]
```

which means "If in an initial state, the probability is less than or equal to 0.5 that eventually the system will be in the state s=0".

Simulation techniques can only generate results for paths from one initial state at a time. Therefore, verifying state based properties of this type with a simulator would be of little use.

However, properties that are specified as `P=?` and `R=?`, i.e. those with the path operator as the root node, return the probability of the path formula for the initial state of the model (or, if there is a set of initial states, returns the first one in a lexicographically ordered set of initial states). Therefore, only properties of this form should be considered for approximate model checking.

3.5.2 The Generic Approximation Algorithm

The *generic approximation algorithm* ($\mathcal{GAA}$) [6] is a simple mechanism that allows monotone properties to be model checked to a particular approximation degree and to a particular level of confidence. The algorithm accepts the parameters:

- $\mathcal{M}$ - The PRISM model;
- ϕ - The monotone property to be checked against $\mathcal{M}$;
- k - The maximum path length;
- ε - The *approximation* parameter;
- δ - The *confidence* parameter.

and produces a result such that (derived from and proven in [6]):

$$Prob\left[|\mathcal{GAA}(\mathcal{M},\phi,k,\varepsilon,\delta) - \mu(\mathcal{M},\phi)| \leqslant \varepsilon\right] \geqslant 1-\delta$$

where $\mu(\mathcal{M},\phi)$ is the actual solution over all paths (i.e. the answer using PRISM's numerical engines). The generic approximation algorithm is given in Figure 3.4:

$\mathcal{GAA}(\mathcal{M},\phi,k,\varepsilon,\delta)$	
1.	$N := 4\log(\frac{2}{\delta})/\varepsilon^2$
2.	$A := 0$
3.	**for** $i = 1$ **to** N **do**
4.	Generate a random path σ of length k with the model $\mathcal{M}$
5.	**if** (ϕ is true on σ) **then** $A := A + 1$
6.	**return** A/N

Figure 3.4: The generic approximation algorithm [6]

The number of iterations required by this algorithm is $\mathcal{O}(\frac{1}{\varepsilon^2}\cdot\log\frac{1}{\delta})$ meaning that the complexity is logarithmic in $1/\delta$, but more crucially quadratic in $1/\varepsilon$. This means that it is possible to choose very small δ values, but if $\varepsilon \leqslant 10^{-2}$, N becomes unreasonably large for realistically sized models.

3.5.3 Extensions to the Generic Approximation Algorithm

A number of extensions are possible to improve the efficiency of the generic approximation algorithm and also to make it applicable to rewards based formulae.

Stopping at Satisfaction, Dissatisfaction and Time Bounds

The generic approximation algorithm can be optimised by only generating paths for as long as they need to be. For example, if a path, σ, satisfies a particular path formula, ϕ, in less than k steps, there is no need to continue the generation of σ until it reaches length k. Also, if a path, σ, can never satisfy a particular path formula, i.e. if the first operator of an "until" formula evaluates to false before the second is satisfied, there is also no need to continue the generation of σ.

Furthermore, if the upper time bound of a bounded path formula is met in less than k steps, i.e. for DTMCs and MDPs, this is simply when the path length is greater than the time bound and for CTMCs this is where the sum of the times spent in each state along the path, $\sum_{k=0}^{i-1} t_k$, is greater than the time bound, there is also no need to continue the generation of σ.

Stopping at the Detection of a Loop

If a path, σ is *looping deterministically* between two states s_q and s_r and the path formula ϕ has not been satisfied by state s_r, then ϕ will never be satisfied over σ. Therefore, there is no need to continue the generation of σ until it reaches length k. A loop is detected when:

- The update event sets for the last l states in σ only have one distinct destination state, i.e.

$$\forall s_{n-j} \in \sigma \mid j = (0..l), \big(|\{s'_{e_i} | e_i \in E_{s_{n-j}}\}| = 1\big)$$

- The update set for the last state in σ, E_{s_n} has one distinct destination state, s' that has appeared at least once in the last l steps.

Modification for Rewards Properties

The generic approximation algorithm was not designed for use with rewards based formulae and so slight modifications are required for each type.

For a"cumulative" formula, ρ, the modification is shown in Figure 3.5.

$\mathcal{GAA}_{\text{cumulative}}(\mathcal{M}, \rho, k, \varepsilon, \delta)$	
1.	$N := 4\log(\frac{2}{\delta})/\varepsilon^2$
2.	$A := 0.0$
3.	**for** $i = 1$ **to** N **do**
4.	Generate a random path σ with the model $\mathcal{M}$, until
5.	the time bound in ρ is met
6.	$A := A +$ reward accumulated forρ over σ
7.	**return** A/N

Figure 3.5: The generic approximation algorithm for cumulative reward based properties

It should be noted that loop detection should not be used for "cumulative" reward formulae. This is because rewards can be accumulated for several loop cycles before the time bound is met.

For an "instantaneous" formulae, ρ, the modification is shown in Figure 3.6.

$\mathcal{GAA}_{\text{instantaneous}}(\mathcal{M}, \rho, k, \varepsilon, \delta)$	
1.	$N := 4\log(\frac{2}{\delta})/\varepsilon^2$
2.	$A := 0.0$
3.	**for** $i = 1$ **to** N **do**
4.	Generate a random path σ with the model $\mathcal{M}$, until:
5.	the time bound in ρ is met in state s_i
6.	$A := A +$ state reward sr_i
7.	**return** A/N

Figure 3.6: The generic approximation algorithm for instantaneous reward based properties

Like "cumulative" reward formulae, for "instantaneous" reward formulae, loop detection should not be used. This is because several loop cycles may be required to meet the time bound and thus calculate the reward at that instant.

For "reachability" reward formulae, ρ of the form `R=? [ F <prop> ]`, if the set of states specified by `<prop>` are unreachable for any path, then the average reward will be infinite. Therefore, if a loop is detected or the maximum path length, k, is reached before `<prop>` is reached, the algorithm can terminate and return ∞. The modifications are shown in Figure 3.7.

$\mathcal{GAA}_{\text{reachability}}(\mathcal{M}, \rho, k, \varepsilon, \delta)$

1. $N := 4\log(\frac{2}{\delta})/\varepsilon^2$
2. $A := 0.0$
3. **for** $i = 1$ **to** N **do**
4. Generate a random path σ with the model $\mathcal{M}$, until either:
5. <prop> is satisfied is some state,
6. a loop is detected, in which case, **return** ∞,
7. the path length, k, is met, in which case, **return** ∞.
8. $A := A +$ reward accumulated forρ over σ
9. **return** A/N

Figure 3.7: The generic approximation algorithm for reachability reward based properties

Verification of Multiple Properties

With a few modifications to $\mathcal{GAA}$, it is possible to verify multiple probability and reward based properties simultaneously. If a function, $known(\phi$ or $\rho, \sigma, i, k)$, is defined to says whether the result of a particular property, ϕ or ρ, is known over a particular path, σ, of length i where $i <= k$:

- For a "bounded until" formula, ϕ, of the form, [<prop1> U<time> <prop2>], the function $known(\phi, \sigma, i, k)$ returns *true* if:
  - $i = k$ and <prop2> has not been satisfied;
  - <prop1> is not satisfied in a state and <prop2> is not satisfied within the time bound <time>;
  - $i < k$, and <prop2> has been satisfied;
  - σ is looping and <prop2> has not yet been satisfied.
- For an "until" formula, ϕ, of the form, [<prop1> U <prop2>], the function $known(\phi, \sigma, i, k)$ returns *true* if:
  - $i = k$ and <prop2> has not been satisfied;
  - <prop1> is not satisfied in a state and <prop2> is satisfied;
  - $i < k$, and <prop2> has been satisfied;
  - σ is looping and <prop2> has not yet been satisfied.
- For a "next" formula, ϕ, of the form, [X <prop>], the function $known(\phi, \sigma, i, k)$ returns *true* if:
  - $i = 1$.
- For "cumulative" and "instantaneous" reward formulae, ρ, with the time bound <t>, the function $known(\rho, \sigma, i, k)$ returns *true* if:
  - $i = k$ and the time bound, <t> has not been met in σ;
  - $i < k$ and the time bound, <t> has been met in σ.
- For a "Reachability" reward formulae, ρ, of the form [R <prop>], the function $known(\rho, \sigma, i, k)$ returns *true* if:
  - $i = k$ and <prop> has not been satisfied;
  - $i < k$, and <prop> has been satisfied;
  - σ is looping and <prop> has not yet been satisfied.

$\mathcal{GAA}_{\text{multiple}}(\mathcal{M}, \phi[], \rho[], k, \varepsilon, \delta)$

1. $N := 4\log(\frac{2}{\delta})/\varepsilon^2$
2. **for each** $\phi \in \phi[]$
3. **define** $A_\phi := 0$
4. **for each** $\rho \in \rho[]$
5. **define** $A_\rho := 0.0$
6. **for** $i = 1$ **to** N **do**
7. Start a path, σ in the initial state.
8. $j := 0$
7. **while** $(\exists \phi \in \phi[] | known(\phi, \sigma, j, k) = \text{false})$ or $(\exists \rho \in \rho[] | known(\rho, \sigma, j, k) = \text{false})$
8. For the current state, s_j automatically sample s_{j+1} with $\mathcal{M}$ and add it to σ
9. $j := j + 1$
10. **for each** $\phi \in \phi[]$
11. if ϕ is satisfied over σ, $A_\phi := A\phi + 1$
12. **for each** $\rho \in \rho[]$
13. $A_\rho :=$ reward accumulated for ρ over σ
14. **return** $(\mathscr{P}, \mathscr{R})$ where:
15. $\mathscr{P} := \{A_\phi/N | \phi \in \phi[]\}$
16. $\mathscr{R} := \{A_\rho/N | \rho \in \rho[]\}$

Figure 3.8: The Generic approximation algorithm for multiple properties

it is possible to modify $\mathcal{GAA}$ as shown in Figure 3.8 to deal with a set, $\phi[]$ of probability based properties and a set, $\rho[]$ of reward based properties.

This algorithm returns a tuple $(\mathscr{P}, \mathscr{R})$ where $\mathscr{P}$ is the set of answers for each property in $\phi[]$ and $\mathscr{R}$ is the set of answers for each property in $\rho[]$.

Simulataneous approximate verification provides a significant improvement in efficiency because the most expensive part of the algorithm is the generation of σ. This now only has to be done N times, rather than N multiplied by the number of properties, as is the case if they are evaluated separately.

Chapter 4

Requirements

This chapter gives an account of the approach towards gathering the requirements for the PRISM simulator. The requirements elicitation process is described first in Section 4.1, before the detailed system requirements for the project are given in Section 4.2.

4.1 Requirements Elicitation

In order to determine the best requirements elicitation strategy, the following factors were taken into account:

- The user who was available for requirements gathering is an expert in the field of probabilistic model checking. He researched, designed and implemented the PRISM tool.
- Other users would become available towards the end of the development cycle because a they were required to use PRISM for a university coursework project.
- There was already a developed PRISM simulator prototype. This was developed to investigate which features would be useful for the user interface as well as how those features should be laid out.
- There was a substantial amount of domain knowledge that needed to be researched before the requirements could be gathered.

It was therefore decided that the first stage of the requirements elicitation process would be to obtain the domain knowledge required to approach this project. This research was performed as a series of meetings with the expert user who also recommended a number of academic texts and books. The result of this research and its analysis was presented in Chapter 3.

With this knowledge, a series of preliminary interviews with the expert user were performed in order to develop the ideas taken from the prototype simulator. From this, most of the requirements for the simulator user interface were developed and drafted as the original requirements document. This document was approved by the expert user and so from this document, the system was designed and developed until it was ready to be given to users for feedback. This feedback allowed a number of usability issues to be resolved.

4.2 Software Specification

This section gives the specification of the PRISM simulator. Because the tool had to be integrated into an existing tool, the first section describes how this integration should be structured. The following sections describe the requirements for the various sub-systems of the tool.

4.2.1 Structural Requirements

R1 The simulator will be structured into three main components: The simulator engine, the simulator user interface and the properties user interface. Figure 4.1 highlights the dependencies of each component. The requirements for each component are described in

requirements R4-R7 (engine), R8-R13 (simulator user interface) and R14-R17 (properties user interface).

R2 The simulator will be integrated into the existing PRISM graphical user interface. In particular, the simulator and property user interface components will be built upon the framework provided by the existing user interface packages and the engine component will be built upon data structures provided by the existing PRISM language parsing package. This is summarised in Figure 4.1.

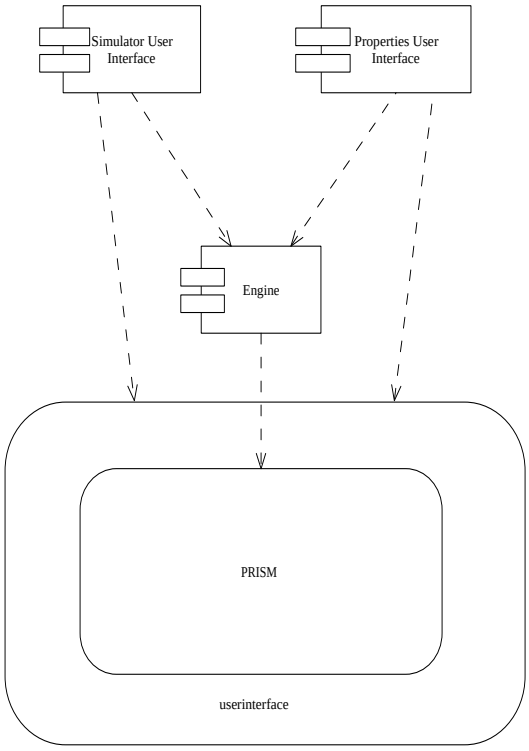


Figure 4.1: PRISM Simulator Package Dependencies.

R3 The PRISM version number that the simulator will be built upon is: PRISM 2.1.
Note: It may also be necessary to upgrade to later versions of PRISM as they are developed. Therefore, the simulator should be written in such a way to make this process as easy as possible.

4.2.2 Engine Requirements

R4 The simulator engine will be able to load one PRISM model at a time. When no PRISM model is loaded, the only function that will be allowed is that of loading a model into the simulator (See R4.3).

R4.1 A loaded PRISM model, will store the following data:

R4.1.1 The model type: $type \in \{notloaded, dtmc, mdp, ctmc\}$.
Rationale: The model type must be stored as some simulator functionality will differ according to its value.

R4.1.2 A set of global variables, G. A variable will be defined as a named integer or Boolean value. Each integer variable will have a fixed minimum, maximum and initial value. Each variable will have a value known as its current state.

R4.1.3 A set, M of modules. Each module, $m_i \in M$ will be as described in background Section 3.2.

R4.2 Initially, no model will be loaded into the engine. Therefore, $type \mapsto notloaded$, $G \mapsto \emptyset$, $U \mapsto \emptyset$ and $M \mapsto \emptyset$.

R4.3 The engine will provide a function that loads a PRISM model. The input to this function will be a ModulesFile object, which is provided by PRISM's parser package. The following prerequisites apply to the ModulesFile object:

- All of its *undefined constants* must have been defined.
- It must be the result of a successful parse of a PRISM model file.

This function will first unload any model that is already loaded and then set *type*, G, and M to be the same as the equivalent data stored in the ModulesFile.

R5 At any particular time, the variables associated with each module in M and the variables in G will be in a particular configuration. This configuration will be known as the *current state*, s_{curr}

R6 The engine will store and manipulate an execution path, σ.

R6.1 The simulation path, σ, will be as defined in Section 3.3.1. i.e. Each state, $m \in \sigma$, will be a non-empty set of variables whose names correspond to the variable names of the global state.

R6.2 The engine will provide a function which starts a new simulation path. The input to this function will be an initial state $\bar{s}$. $\bar{s}$ will be a set of variables whose names correspond to the variable names of in s_{curr}. Firstly, the function assigns: $s_{curr} = \bar{s}$. s_{curr} then appended to σ as s_0, as in R6.7. *Rationale: When a simulation path is started, it should be possible to start from any initial state.*

R6.3 If *type* $\mapsto$ *ctmc*, σ will also be associated with a variable, t_σ, which represents how much time has elapsed in the execution of the path so far. When any new execution path is started, $t_\sigma \mapsto 0$.

R6.4 σ will be associated with a variable $\mathcal{SR}_\sigma$ which represents the total state reward accumulated over the execution path so far. This is calculated as $\sum_{i=0}^{k} sr_i \cdot t_i$ where k is the length of σ. When any new execution path is started, $\mathcal{SR}_\sigma \mapsto 0$.

R6.5 σ will be associated with a variable $\mathcal{TR}_\sigma$ which represents the total transition reward accumulated over the execution path so far. This is calculated as $\sum_{i=0}^{k} tr_i$ where k is the length of σ. When any new execution path is started, $\mathcal{TR}_\sigma \mapsto 0$.

R6.6 σ will be associated with a variable, $\mathcal{R}_\sigma$ which represents the total reward accumulated over the execution path so far. This is calculated as $\mathcal{SR}_\sigma + \mathcal{TR}_\sigma$.

R6.7 The engine will provide a function to append a copy of s_{curr} to the execution path, σ.

R6.8 The engine will provide a function which removes all states from σ except for s_0. This is known as *resetting the path.*

R6.9 The engine will provide a function which, for a given index, i, removes all states, $s_j \in \sigma$, where $j > i$. This is known as *backtracking.*

R6.10 The engine will provide a function which, for a given index, i, removes all states, $s_j \in \sigma$, where $j < i$. Furthermore, i should the index for each remaining state, transition and reward in σ. i.e. the state, s_i will become s_0. This is known as *removing preceding states.*

R6.11 For any current state, s_{curr}, the simulator will be able to provide a list, $E_{s_{curr}}$, of possible update events. The calculation of $E_{s_{curr}}$ will be as described in Section 3.3.2 for when *type* $\mapsto$ *dtmc*, Section 3.3.3 for when *type* $\mapsto$ *mdp* and Section 3.3.4 for when *type* $\mapsto$ *ctmc*.

R6.12 It will be possible to determine whether σ is *looping.* A loop is detected when:

- The update event sets for the last k states in σ only have one distinct destination state, i.e.
$$\forall s_{n-j} \in \sigma \mid j = (0..l), \left(|\{s'_{e_i} | e_i \in E_{s_{n-j}}\}| = 1\right)$$
- The update set for the last state in σ, E_{s_n} has one distinct destination state, s' that has appeared in the last k steps.

It will be possible to request the value of $n - k$, i.e. to ask where the looping section begins.

R6.13 The engine will provide a function that for a given list of update events, E_s, and an index i, selects the ith event, e_i of E_s, and then:

- Sets $s_{curr} = s'_{e_i}$;
- Appends s_{curr} to σ as in R6.7.

If $type \mapsto ctmc$, this function can also accept a time parameter, t. If this parameter is provided, t and if $s_i = s_{curr}$ then t_{i-1} is set to t. If no parameter is provided, the value for t is calculated automatically using Equation 3.2.
Rationale: The engine should provide a function to cater for *manual* event choices by users.

R6.14 The engine will provide a function that given an input value, n, will make n automatic update event choices. For each iteration:

- The current set of update events, $E_{s_{curr}}$ is calculated for the current state, s_{curr}. If $E_{s_{curr}} \mapsto \emptyset$, s_{curr} is a *deadlock* state and so the iterations should stop. Also, if a loop is detected, as in R6.12, iterations should stop.
- The simulator engine should then sample an event, e from $E_{s_{curr}}$. This should be done as described in Section 3.3.2 if $type \mapsto dtmc$, Section 3.3.3 if $type \mapsto mdp$ and Section 3.3.4 if $type \mapsto ctmc$.
- Once an event, e has been sampled:
  - Set $s_{curr} = s'_e$;
  - Append s_{curr} to σ as in R6.7.

R6.15 The simulator engine will provide a function that determines whether, for any Boolean predicate, ϕ, and index, i, $\phi \xrightarrow{evaluates}$ true in the state $s_i \in \sigma$.

R6.16 The simulator engine will provide a function that determines whether for a given index, i, the state $s_i \in \sigma$ is:

R6.16.1 A deadlock state. i.e. The event update set for s_i is empty, i.e. $E_{s_i} = \emptyset$. -or-

R6.16.2 The initial state. i.e. whether $s_i = s_0$.

R6.17 The engine will provide a function which, for a given path formula, states whether that formula is true over the current execution path, σ. Path formulae are defined and their results calculated as in Section 3.4.

R6.18 The engine will provide a function, which for a given rewards formula, states what the calculated reward is over the current execution path, σ. Rewards formulae are defined and their results calculated as in Section 3.4.

R7 For any set of PRISM path or rewards properties, the simulator engine will provide a function which performs approximate verification of those properties over the current model. This function will use the sampling algorithm, $\mathcal{GAA}_{\text{multiple}}$, defined in Figure 3.8. As well as the properties itself, the function will require as input, either:

- The number of iterations for the sampling algorithm;
- An approximation parameter, ε, and a confidence parameter, δ. The number of iterations is calculated using the formulae:

$$\frac{4 \log_{10}(\frac{2}{\delta})}{\varepsilon^2}$$

It will also require the maximum path length, k and an initial state, $\bar{s}$ for each sample path, $\bar{s}$. The results returned will be set of probabilities (for path properties) and positive real numbers (for rewards properties).

4.2.3 Simulator User Interface Requirements

R8 **General Requirements**

R8.1 The simulator user interface (SUI) will be contained within a new tab panel in the existing PRISM graphical user interface.
Rationale: The functionality provided by the simulator should not interfere with the existing features of the PRISM GUI. Displaying the SUI in a tab panel is also consistent with the rest of the PRISM GUI, where groups of functions (i.e. editing, verification, log) are separated by tabs.

R8.2 At the highest level, the SUI will be in one of 3 states:

No model When no valid model has been parsed via the 'model' tab of the PRISM GUI, the SUI will be disabled. The SUI will return to this state only when the user has selected 'new' model, or when an invalid (one that cannot be parsed) model is loaded.

Model parsed, no path When a valid PRISM model has just been parsed, the SUI will be in this state. To move into this state, the simulator engine will have to load the parsed PRISM model, as described in R4.3 and will have to remove the current execution path, σ from the simulator engine.

Model parsed, path active When an execution path is started, as in R6.2, the simulator will move into this state, and remain in this state whilst any model exploration takes place.

Rationale: The simulator should always work with the model that is displayed in the parse tree of the 'model' tab of the PRISM GUI (See the PRISM manual for more details [2]). When the model is edited and thus reparsed, any simulation results would be invalid for the current model and so the simulator must be reset.

R8.3 The simulator user interface will display the following user interface components:

Execution Path This will be a table that provides a view of the current execution path, σ, defined in R6.1. Each row will show the following information about each model state in σ:

- The index of the model state;
- The values of each variable of the model state;
- If $type \mapsto ctmc$, the time spent in the model state;
- The reward associated with the model state;
- The reward associated with the transition away from the model state;

This table will only be enabled if the SUI is in the (Model parsed, path active) state.

Path Information The following statistical information about the current execution path, σ, will be displayed:

- Model Type - Whether the model is a DTMC, MDP or a CTMC;
- Path Length - The current length of σ;
- Total Time - The sum of transition times, t_σ, as defined in R6.3;
- State Rewards - The state reward accumulated, $\mathcal{SR}_\sigma$, as defined in R6.4;
- Transition Rewards - The transition reward accumulated, $\mathcal{TR}_\sigma$, as defined in R6.5;
- Total Reward - The total reward accumulated, $\mathcal{R}_\sigma$, as defined in R6.6;

When the SUI is in the (No model) state, each of the above will display dashes, '-'. When the SUI is in the (Model parsed, no path) state, only the 'Model Type' field will be enabled. When the SUI is in the (Model parsed, path active) state, every field is displayed with appropriate values.

Path Control Panel The following buttons will be provided:

- New Path;
- Reset Path;
- Export Path.

When the SUI is in the (No Model) state, these buttons will be disabled. In the (Model parsed, no path) state only the 'New Path' button will be enabled. In the (Model parsed, path active) state all of these buttons will be enabled.

Update Event Table This will be a table that has the ability to display sets of event updates, E_s for a particular state, s. Each row will display information about each event, $e \in E_s$:

- The probability of the event, $\mathbf{P}_e$ when $type \mapsto dtmc$ or $type \mapsto mdp$;
- The rate of the event, $\mathbf{R}_e$ when $type \mapsto ctmc$;
- The action label of the event, A_e, enclosed within square brackets. If $A_e \mapsto$ no action label, only "[]" will be displayed.
- The name of the module, M_e, of the event.
- A description of the assignments that are required to update s to s'_e.

This table will only be enabled if the SUI is in the (Model parsed, path active) state.

Model Exploration Panel The following buttons will be provided:

- Automatic Update - This will be accompanied by a text field named, 'No. Steps'.
- Manual Update - For CTMCs this will be accompanied by a text field named, 'Time'.
- Backtrack - This will be accompanied by a text field named, 'To Step'.
- Remove Preceding - This will be accompanied by a text field named, 'To Step'.

These buttons will only be enabled if the SUI is in the (Model parsed, path active) state.

Path Formulae List This will be a list of path formulae with icons. For each formula, the icon can either be:

- A question mark;
- A tick;
- A cross;
- A number symbol.

State Formulae List This will be a list of state label formulae with icons. For each formula, the icon can either be:

- A cross;
- A tick.

R9 **Execution Path Table Requirements** - These requirements describe the functionality that will be provided concerning execution path table, defined in R8.3.

R9.1 It will be possible to adjust how the execution path table's columns and headers are displayed. In particular:

- It will be possible to resize each column;
- It will be possible to hide, and re-display each column.

Rationale: It is very likely that a lot of information will be produced for each execution path. It is likely that the user will not be interested in a lot of this information, or will be more interested in certain pieces of information. Providing this flexibility will allow users to view whatever information they wish, in whichever order.

R9.2 At any point, the last row in of the execution path table will display the current state of the model. By default, this row will be *selected*. The selected row will be coloured differently to all other rows. It will also be possible to select other rows, one at a time. If for any reason, a row is unselected (by pressing ctrl-click), the last row in the table will get re-selected.

R9.3 The set displayed in the update events table, defined in R8.3, will display the update event set, E_{s_i} for the state, s_i displayed by the selected row of the execution path table. This is calculated by the simulator engine as defined in R6.11 If $s_i \neq s_{curr}$, the

row corresponding to the event that was selected to get into $s_{(i+1)}$ will be selected in the update events table. If $s_i = s_{curr}$ the update events table will allow the user to select one of the update events by clicking on the corresponding row in the table.

R9.4 When the 'New Path' button is selected, so long as the SUI is in the (Model parsed, no path) or in the (Model parsed, path active) state, the following will happen:

- The user will be prompted to define all undefined constants for the current PRISM model. There will be a dialog containing fields for each undefined constant name. If a value has already been entered for a previous 'New Path', by default, this value will be given.
  After the user has entered the values they require, they can either press an 'Okay' button, or a 'Cancel' button. When the 'Okay' button is pressed, each undefined constant field will be checked. For integer constants, this validity check will depend on whether the entered string is a correctly formed integer. For real constants, this validity check will depend on whether the entered string is a correctly formed double, as defined in Java's Double.parseDouble method. For Boolean variables, the strings "true", "false" and any correctly formed integer are accepted. For integers, '0' is treated as false and any other value is treated as true. if the 'Cancel' button is pressed at any time, the whole 'New Path' action is cancelled.
  After the validity check, the defined constants will be loaded into the current PRISM ModulesFile.
- The user will be prompted to define an initial state, $\bar{s}$. This will be a dialog containing fields for each variable name of the global state. Each field will contain a default value calculated using the 'GetDefaultInitValues' method of the ModulesFile.
  After the user has entered the values they require, they can either press an 'Okay' button or a 'Cancel' button. When the 'Okay' button is pressed, each variable field will be checked. The validity checks will be the same as for the undefined constants dialog, except that there will be no real-number variables. If the 'Cancel' button is pressed at any time, the whole 'New Path' action is cancelled.
- The current PRISM ModulesFile (with constants defined), will then be loaded into the simulator engine, as defined in R4.3.
- The simulator engine will then be requested to start a new execution path, using the function defined in R6.2, with $\bar{s}$ as a parameter. As described in R8.2, the SUI will move into the (Model parsed, path active state). The execution path table and update events table will update themselves to reflect the simulator engine's new execution path, σ.

R9.5 When the 'Reset Path' button is selected, the simulator engine will *reset the execution path*, as described in R6.8.

R9.6 When the 'Export Path' button is selected, the following will happen:

- The user will be prompted to enter a name of a new file. The will be done using a standard file-chooser, with the default file extension set to 'txt'.
- Once the file has been selected, the current simulation path table will be written to it in the following way:
  - The first line will contain a space separated list of all displayable column headers from the simulation path table.
    Note: Even columns that have been hidden will be outputted.
  - Each subsequent line will contain a space separated list of the data displayed in each row of the simulation path table.

R9.7 When the 'Backtrack' button is selected, the simulator engine will remove all states that follow the model state indexed by the value entered in the 'To State' field. If this is not a valid positive (including 0) integer that is less than the execution path's length, an error will be given. If the value is valid, the simulator engine's backtracking function, as described in R6.9, should be used with the value as a parameter.

R9.8 When the 'Remove Preceding' button is selected, the simulator engine will remove all states that precede the model state indexed by the value entered in the 'From

State' field. If this is not a valid positive (including 0) integer that is less than the execution path's length, an error will be given. If the value is valid, the simulator engine's remove preceding states function, as described in R6.10, should be used with the value as a parameter.

R9.9 If the simulator engine has detected a loop in the current execution path, σ, an arrow from the last row in the execution path table to where the loop starts should be displayed. The querying of the simulator engine for the presence of a loop and for when the loop starts is described in R6.12.

R10 When the 'Manual Update' button is selected, the simulator engine will perform a 'manual' update described by the event that is currently selected in the update events table; this is described in R6.13. If $type \mapsto ctmc$, and if the 'Auto-time' checkbox is not selected, the value described in the 'Time' field will be provided as a parameter to the simulator engine function. If the value is not a valid positive real number (> 0.0), an error will be given.

R11 When the 'Automatic Update' button is selected, the simulator engine will make n automatic update event choices, as described in R6.14. n is obtained from the value in the 'No Steps' field. If the value is not a valid positive integer an error will be given.

R12 **State Formulae List requirements** - These requirements describe the functionality that will be provided concerning state formulae list, defined in R8.3.

R12.1 The state formulae list will be populated with valid PRISM formulae labels when the "New Path" button is selected. These formulae will be taken from the labels of the current properties file loaded into the PRISM graphical user interface. Only the name of the label will be displayed in the list. Furthermore, the list will always contain a label named "init" and a label named "deadlock".

R12.2 If a formula in the state formulae list is satisfied in the *selected* state, it will display a tick icon. If not, it will display a cross icon. Satisfaction of a formula over a particular state was described in R6.15. For the "init" and "deadlock" labels, this was described in R6.16

R12.3 It will be possible to select one formula from the state formulae list at a time. When a formula is selected, all states in the execution path table that satisfy the formula will be highlighted in a different colour.

R13 **Path Formulae List Requirements** - These requirements describe the functionality that will be provided concerning Path Formulae List, defined in R8.3.

R13.1 The Path Formulae List will be populated with valid path and rewards formulae when the "New Path" button is selected. These formulae will be taken from the simulatable (see Section 3.5.1) properties of the current properties file loaded into the PRISM graphical user interface. Only the path or reward part of the property will be taken, i.e. if the property is `P=? [true U s=4]` only `true U s=4` will be displayed. Furthermore, duplicate formulae will be removed.

R13.2 If the result of a formula cannot be determined over the current execution path σ, the question icon will be displayed; If the formula is a path formula, and is proven to be satisfied over σ, the tick icon will be displayed; If the formula is a path formula, and can not be satisfied over σ, the cross icon will be displayed; If the formula is a reward formula, and the reward has been calculated over σ, the numeric icon will be displayed.

R13.3 If the result of a formula has been determined, and the user hovers the mouse over the the corresponding element of the list, a tooltip will be shown that displays the result.

R13.4 It will not be possible to select elements of the Path Formulae List with the mouse.

4.2.4 Properties User Interface Integration Requirements

R14 In the properties user interface there is a list of actions for functions on properties that have been loaded into the PRISM user interface. The list of actions is displayed as both a popup menu and a menu on the title bar. A button labelled "Simulate" will be added to this list of actions.

R15 In the properties user interface, it is possible to select a set of properties from those that have been loaded. If at least one of those properties can be verified by the PRISM simulator, (see Section 3.5.1 for a description of which properties can/cannot be simulated), the "Simulate" action is enabled.

R16 If the "Simulate" action is selected by the user, the following will occur:

- The user will be prompted to define all undefined constants for the current PRISM model. This will be a dialog containing fields for each undefined constant name. If a value has already been entered for a previous 'New Path', by default, this value will be given.

  After the user has entered the values they require, they can either press an 'Okay' button, or a 'Cancel' button. When the 'Okay' button is pressed, each undefined constant field will be checked. For integer constants, this validity check will depend on whether the entered string is a correctly formed integer. For real constants, this validity check will depend on whether the entered string is a correctly formed double, as defined in Java's Double.parseDouble method. For Boolean variables, the strings "true", "false" and any correctly formed integer are accepted. For integers, '0' is treated as false and any other value is treated as true. if the 'Cancel' button is pressed at any time, the whole 'New Path' action is cancelled.

  After the validity check, the defined constants will be loaded into the current PRISM ModulesFile.

- The user will be prompted with a dialog, entitled "Approximate Verification Parameters", "to define an initial state, $\bar{s}$. This will be a dialog containing fields for each variable name of the global state. Each field will contain a default value calculated using the 'GetDefaultInitValues' method of the ModulesFile.

  The same dialog will enable the user to define the approximation parameter, ε and the confidence parameter, δ, as well as the maximum path length k.

  After the user has entered the values they require, they can either press an 'Okay' button or a 'Cancel' button. When the 'Okay' button is pressed, each variable field will be checked. The validity checks will be the same as for the undefined constants dialog, except that for the initial state and k there will be no real-number values allowed. If the 'Cancel' button is pressed at any time, the whole "Simulate" action is cancelled.

- The current PRISM ModulesFile (with constants defined), will then be loaded into the simulator engine, as defined in R4.3.

- The simulator engine will then be requested perform approximate verification of all of the simulatable (see Section 3.5.1) properties using the function described in R7 with $\bar{s}$, k, δ and ϵ as parameters.

- It will also be possible to request that each property be verified separately. Therefore, the function described in R7 will be called once for each property, again with $\bar{s}$, k, δ and ϵ as parameters.

- The results will be passed to the properties user interface, which already provides the means of displaying those results to the user.

R17 In order to run an *experiment* with the simulator, it will be necessary to integrate the approximate verification function, described in R7, into the existing property experiments mechanism:

R17.1 The existing dialog that requests that users define which constants they wish to vary for the experiment will be modified to include a checkbox. A user will be able to select this checkbox entitled "Use Simulation?" if they wish to use the simulator for their experiment.

R17.2 If the "Use Simulation?" checkbox is selected, the user will be prompted with the "Approximate Verification Parameters" dialog, described in R16.

R17.3 For each configuration of model constants, those constants will be loaded into the current PRISM ModulesFile, which will then be loaded into the simulator engine, as defined in R4.3. Each configuration of property constants will be enumerated as a set of properties. The simulator engine will then be requested to perform approximate verification fo these properties using the function described in R7 with the $\bar{s}$, k, δ and *epsilon* collected from the "Approximate Verification Parameters" dialog.

R17.4 The results collected will be passed to the existing mechanism for dealing with experiments in the properties user interface.

Chapter 5

Design and Implementation

This chapter describes the design for the PRISM simulator. It begins by giving a high level system architecture, detailing the responsibilities of each component. The focus of this chapter is on the design of the simulator engine. This is where all of the core data structures and algorithms for this project are contained. This is discussed in detail in Section 5.2. Some of the key user interface design choices are then outlined in Section 5.3. Finally, the main implementation issues are outlined in Section 5.4.

5.1 System Architecture

At the highest level, the architecture of the system is in three components:

- The simulator engine - this component is responsible for providing all data structures and algorithms for path generation and manipulation and for performing statistical sampling techniques. It should conform to the interface given in the Specification;
- The simulator user interface - this component is responsible for providing an intuitive view of the execution path and for providing ways for users to manipulate it visually. The simulator user interface relys on functions provided by the simulator engine.
- Integration into PRISMs properties user interface - this component is responsible for providing a means of running approximate model checking techniques from the existing PRISM user interface. Like the simulator engine, this component relys on the functions provided by the simulator engine.

5.2 Simulator Engine Design

The simulator engine provides all of the core functionality of the PRISM simulator. The structure of the engine is as follows:

- Storage of the current state;
- Storage of the current model;
- Storage of the current execution path through the current model;
- The update handler module, responsible for calculating update events for a particular state, handling manual choices from those sets and selecting updates automatically from those sets;
- The loop detection module which looks at both the current execution path and the update handler;
- The path evalution module investigates whether formula are satisfied in the current execution path;
- The sampling module, which performs approximate model checking.

This is summarised in the Figure 5.1.

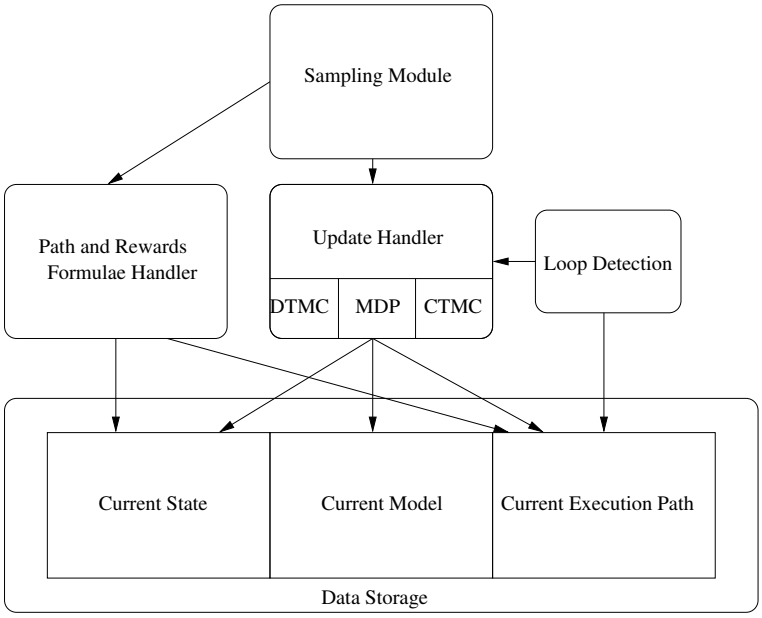


Figure 5.1: Structure of the simulator engine

5.2.1 Model Representation

One of the main data structures that had to be considered was the storage of the current PRISM model. PRISM itself takes a PRISM model file and *parses* it into an abstract syntax tree structure. Previous simulator implementations, including PrismSim (Section 2.1) and the PRISM simulator prototype (Section 2.2) used this abstract syntax tree structure directly.

Although it required extra time and effort, it was decided that a new data structure would be designed specifically for the simulator engine. This was to address the main weakness in the PRISM model abstract syntax tree structure, which was that it was designed for the purposes of parsing PRISM model files, and not for direct calculation. The following issues were noted:

- Querying of variables was always done by searching for the value by the name of the variable;
- Expression evaluate requires lots of checking what the type of a particular object is (instanceof checks);
- In order to evaluate the value of expressions, an array of constants has to be passed around as a parameter.

The Current State

In order to address the issue of variable querying, it was decided that the state of the current model, s_{curr}, would be stored as an integer array called `model_variables` of length `num_model_variables`. Therefore, references to variables would not be done using the variable's name, but the variable's index in `model_variables`.

Expressions

The referencing of variables is always done by *expressions*. Expressions are key to the representation of the current model, but also for other parts of the PRISM engine; they are used for:

- The guards of each command;
- The probabilities or rates of each update in each command;
- The result of a variable assignment;

- State propositions;
- State and transition rewards.

It was important that the storage of expressions in the simulator engine was consistent with the storage of expressions in the PRISM model abstract syntax tree. Therefore, the following features had to be provided (based on the description given in the PRISM manual [2]):

- Literal values, e.g. `1.54`, `5`, `true`, `false`, etc.;
- Variable names, e.g. `s`, `d`, `queue_size`, etc.[1];
- Multiplication and division, e.g. `d * 2`, `1 / (2+3)`, etc.;
- Addition and subtraction, e.g. `s + 4`, `10 - s`, etc.;
- Relational operators, e.g. `<`, `<=`, `>=`, `>`, `=`, `!=`[2];
- Boolean *not* operator, e.g. `!flag`;
- N-ary Boolean operators, *and* and *or*, e.g. `a & b & c`, `a | b | c`, etc.;
- Conditional evaluation, `<cond>?<a>:<b>` meaning if `<cond>` is true, then `<a>` else `<b>`, e.g. `a<10?true:false`;
- N-ary functions, *min* and *max*, e.g. `min(x,y,z)`, `max(x,y,z)`, etc.;
- Binary functions, *pow* and *mod*, e.g. `pow(x, 2)`, `mod(y, 2)`, etc.;
- Unary functions, *ceil* and *floor*, e.g. `ceil(2.88)`, `floor(2.54+a)`, etc.;

The following considerations were taken into account when designing the data structure for storing these expressions:

- Each expression is associated with a type: 'boolean', 'integer' or 'double'.
- One of most common operations in many of the simulator engine algorithms is the `evaluate` function. Therefore, this function should not perform any unnecessary calculations, in particular: there should never be a case whether the type of an object is not known prior to evaluation;

The chosen expression data structure is tree-based, with each node using inheritance to *extend* the `Expression` class. `Expression` is an abstract class that requires that subclasses define at least one of the following methods:

- `int Evaluate(int[]vars)` - This method should calculate the integer value of the expression using the state described by the variable values in the `vars` parameter. This method will only be called if the type of the subclass is 'integer' or 'Boolean'.
- `double Evaluate_Double(int[]vars)` - This method should calculate the value of the expression using the state described by the variables values in the `vars` parameter. This method will only be called if the type of the subclass is 'double'.

The `Expression` class actually defines implementations of `Evaluate` and `Evaluate_Double`. The implementation for `Evaluate` simply casts the result of `Evaluate_Double` to an integer and vice-versa for the `Evaluate_Double` method. Subclasses only have to override the most appropriate method for the expression type.

[1] It should be noted that, unlike the PRISM abstract syntax tree, there is no need to store identifiers such as constants and `formula` constructs. This is because the simulator engine will only ever need to deal with PRISM models that have had all their constants defined and so it is only necessary to store the value. Also, PRISM provides a function that removes all `formula` constructs and replaces them with the expanded expressions.

[2] It should also be noted that there is no need to support the *range* shorthand provided by the PRISM language. This is because an expression such as: `x = 1..4, 5, 9..13` can just be represented as: `(x>=1 & x<=4) | (x=5) | (x>=9 & x<=12)`

The reason for two evaluation method is for efficiency, taking into account that operations on integers are faster than operations on floating-point numbers. An illustrative example is the expression for multiplication. A multiplication expression of the form `expr1 * expr2` has two child expressions. If one of those children evaluates to a real number, then the type of the multiplication expression will also be real. Therefore, the multiplication expression would be of class, `RealTimes`, which itself uses floating-point arithmetic and overrides the `Evaluate_Double` method. This method simply calls the `Evaluate_Double` methods of both of the children and returns the product of the two answers.

However, if both children evaluate to integers, then the type of the multiplication expression will also be integer. Therefore, the multiplication expression would be of class, `NormalTimes`, which itself uses only integer arithmetic and overrides `Evaluate` method. This method simply calls the `Evaluate` methods of both of the children and returns the product of the two answers.

The complete inheritance hierarchy is shown in Appendix B as Figure B.2.

The Command Table

In order to store the commands for the model, it was decided to simply store a single array of `Command` objects. This array, `commands`, is known as the *command table*.

It was important to ensure that each `Command` object stores all of the appropriate information, i.e:

- The module the command belongs to. This is stored as an index, `module_index` where each module has a unique index and module indices are ordered according to occurance in the PRISM model file.
- The action label of the command. This is also stored as an index, `action_index` where each action label has a unique index. Action indices are ordered according to occurance in the PRISM model file. If there is no action label for the command, the value `-1` is stored.
- The command's *guard*. This is stored as a `NormalExpression` object, `guard`, which should evaluate to a Boolean value on any configuration of `model_variables`.
- An array of *updates*. This is stored as an array, `updates`, of `Update` objects. Each `Update` object must store the appropriate information, i.e:
  - The update will be associated with either a probability or a rate. In either case this can be stored as a `RealExpression` object, `prob`, which should evaluate to a real number on any configuration of `model_variables`.
  - An array of *assignments*. This is stored as an array, `assignments`, of `Assignment` objects. Each `Assignment` object must store the appropriate information, i.e:
    - The *index* of the variable in `model_variables` that this assignment object is assigning to. This is stored as an integer, `var_index`.
    - The expression for the assignment itself. This is stored as a `NormalExpression` object, `assign`, which should evaluate to either a Boolean or an Integer value on any configuration of `model_variables` according to the type of the variable being indexed by `var_index`.

For the purposes of synchronisation, it is necessary to store a table known as the *alphabet table*. It is used to find out which modules contain which action labels. This table is stored as a 2-dimensional array, `alphabet` of Boolean values where the first dimension index a module and the second is used to index an action label.

If the ith module contains the jth action label then `alphabet[i][j] == true`.

The Reward Tables

The final component of a PRISM model that needs to be stored is the *rewards construct*. This will be stored as two arrays, one for state reward items, `state_rewards_table` and the other for transition reward items, `transition_rewards_table`.

A state reward item is stored as a `StateReward` object. Each `StateReward` object must store the appropriate information, i.e.

- The reward item's *guard*. This is stored as a `NormalExpression` object, `guard`, which should evaluate to a Boolean value on any configuration of `model_variables`.
- The reward item's *reward value*. This is stored as an `Expression` object, `reward` which will evaluate to the value of this reward on any configuration of `model_variables`.

A transition reward item is stored as a `TransitionReward` object. Each `TransitionReward` object must store the appropriate information, i.e.

- The reward item's *guard*. This is stored as a `NormalExpression` object, `guard`, which should evaluate to a Boolean value on any configuration of `model_variables`.
- The action label of the reward item. This is stored as an index, `action_index`. If there is no action label for the reward item, the value `-1` is stored.
- The reward item's *reward value*. This is stored as an `Expression` object, `reward` which will evaluate to the value of this reward on any configuration of `model_variables`.

Important Functions

Once the model is loaded into memory, it remains static until it is unloaded. Therefore, the only functions that are necessary are those which load and unload the model. These are summarised below:

- For each subclass type of `NormalExpression` and `RealExpression` there is a function which creates a new expression object with appropriate input parameters. For example, to construct a `VariableExpression`, the only input parameter would be the variable's index. For objects such as `NormalPlus`, the input parameters would be references to two `NormalExpression` objects.
- A function is provided to create a new `Assignment` object. This function will require a parameter for each member of the new object. i.e. an index for the `var_index` member and a `NormalExpression` object for the `assign` member.
- A function is provided to create a new `Update` object. This function will require a parameter for each member, i.e. a `RealExpression` object for `prob` and an array of `Assignment` objects for `assignments`.
- A function is provided to create a new `Command` object. This function will require a parameter for each member, i.e. a `NormalExpression` object for `guard`, an index for `action_index` and an array of `Update` objects for `uddates`.
- A function is provided to set up the command table, `commands`. This function will assign to `commands` an input parameter which contains an array of `Command` objects.
- A function is provided to create a new `StateReward` object. This function will require a parameter for each member.
- A function is provided to create a new `TransitionReward` object. This function will require a parameter for each member.
- A function is provided to set up the reward tables. This function will require an array of `StateReward` objects to assign to `state_rewards_table` and an array of `TransitionReward` objects to assign to `transition_rewards_table`.
- A function is provided which sets up the storage of the current state. This will set up the array, `model_variables` to contain the number of variable values specified by an input parameter.
- A function is provided which unloads all model storage information from memory.

There are also a few functions which are provided for calculations concerning the current configuration of `model_variables`. These are summarised below:

- A function, `Calculate_State_Reward`, is provided to calculate the reward for being in a state represented by an input parameter, `variables`, which is an `int` array of the same size as `model_variables`. The reward is calculated by the following algorithm:

```
real Calculate_State_Reward(int[]variables)
1.  R := 0
2.  for each StateReward sr in state_rewards_table
3.      if sr.guard.Evaluate(variables) then
4.          R := R + sr.reward.Evaluate(variables)
5.  return R
```

- A function, `Calculate_Transition_Reward`, is provide to calculate the reward for a transition from a state given by an input parameter, `variables`, with an action label, given as an index parameter, `action_index`. The reward is calculated by the following algorithm:

```
real Calculate_Transition_Reward(int[]variables, int action_index)
1.  R := 0
2.  for each TransitionReward sr in transition_rewards_table
3.      if sr.guard.Evaluate(variables) and
4.          sr.action_index = action_index then
4.              R := R + sr.reward.Evaluate(variables)
5.  return R
```

5.2.2 Path Representation

Much of the key requirements of the simulator engine involve the storage and manipulation of the execution path, σ. In order to design the storage representation of the execution path, a few key decisions had to be made.

Firstly, it had to be decided whether the path should be grow dynamically or whether it should be static. It was decided that a static approach would be best because:

- Any analysis of paths always has a maximum path length, k and there would be nothing gained from evaluating paths passed this point.
- Memory allocation and deallocation are computationally expensive. A static approach allows the memory to be allocated once for each model, or for each time the user wishes to change k wheras a dynamic approach would require a lot of memory allocation and deallocation.

Therefore, it was decided that the path would be stored as an array, `the_path`, of length `k` of `PathState` objects. A `PathState` object provides storage for all information about a particular state in the execution through the current model:

- The configuration of variables for that state. This is stored as an `int` array, `variables`, which is the same length as `model_variables`.
- The state reward. This is stored as a real number, `state_reward`. If the state reward has not yet been calculated for this, the value is stored as -1.
- The reward of the transition from that state. This is stored as a real number, `transition_reward`. If the transition reward has not yet been calculated for this state the value is stored as -1.
- The time spent in that state. This is stored as a real number, `time_in_state`. If the time in this state has not yet been calculated, the value is stored as -1. For discrete-time Markov chain and continuous-time Markov chain models, this value will always be stored as 1.
- The reward of the execution path up until that state. This is stored as a real number, `path_reward`. This is stored to reduce the amount of times it has to be calculated.
- For each state, there is always (apart from deadlock states) a list of possible update events. The index of the event in the list of possible update events is stored as an index, `choice_made`.

Preallocation of the execution path creates k `PathState` objects. The values for each member only need to store junk data at this point.

The current execution path is represented simply by storing an integer, `current_index`, which says the first `current_index` states in `the_path` are actually the states of interest and the states `current_index` ... k are junk data (old execution paths or dummy values).

Therefore, to restart a path, it is only necessary to set `current_index` to -1, rather than deallocate the memory.

Important Functions

Three important functions are necessary for modification of `the_path`:

- A function, `Add_State`, is provided which adds the current state (what is in `model_variables`) to `the_path`. The function accepts two parameters:
  - A real number, `time_in_last_state` which is the time spent in the last state before moving to the state now stored in `model_variables`.
  - An integer, `choice_index` which is the index of the update event that was selected to move into the state now stored in `model_variables`.
  - An integer, `action_index` which is the index of the action that was scheduled by the update event.

  The function does the following:
  - Determines whether `current_index + 1` is greater than the maximum path length. If so, an error is thrown;
  - Increments `current_index`;
  - Copies `model_variables` into `the_path[current_index].variables`;
  - Adds information to the last state, i.e (unless `current_index` is 0):
    * `the_path[current_index-1].time_in_state = time_in_last_state;`
    * `the_path[current_index-1].choice_made = choice_index;`
    * `the_path[current_index-1].state_reward = Calculate_State_Reward (the_path[current_index-1].variables);`
    * `the_path[current_index-1].transition_reward = Calculate_Transition_Reward (the_path[current_index-1].variables, action_index)`
    * `the_path[current_index-1].path_reward = the_path[current_index-2].path_reward + the_path[current_index-1].state_reward * the_path[current_index-1].time_in_state + the_path[current_index-1].transition_reward` unless `current_index` is 1 where the value of the`path[current_index-2].path_reward` is 0.

- A function, `Backtrack`, is provided which allowing backtracking. The function accepts an index parameter, `to_state` which describes where to backtrack to and does the following:
  - Sets `current_index` to be `to_state`;
  - Copies the array stored in `the_path[current_index]` into `model_variables`.

- A function, `Remove_Preceding`, is provided which allows states to be removed before a particular index. The function acepts an index parameter, `before_state` and does the following:

```
Remove_Preceding(int before_state)
1.  i := before_state
2.  j := 0
2.  for i to current_index
3.      the_path[j] = copy of the_path[j]
```

5.2.3 Update Event Set Calculation

One of the key parts of the simulator engine design was the algorithm to calculate the set, E_s, of events from a particular state, s.

Storage of the Event Set

For the same reasons why the execution path, `the_path` is stored as a static preallocated array, it was decided that the update event set, `updates`, would be preallocated too. Therefore, `updates` is stored as an array of `UpdateEvent` objects which have to store the appropriate information:

- The probability or rate of the update event. This is stored as a real number, `prob`;
- The index of the probability distribution of the update event. This is irrelevent for discrete-time Markov chains and continuous-time Markov chains. It is stored as an index, `dist_index`.
- The action index of the update event. This is stored as an index, `action_index`.
- The module index of the update event. This is stored as an index, `module_index`.
- A preallocated array, `assignments` of references to `Assignment` objects. These references point to assignments stored in the commands of the command table. Initially, this array consist of dummy references, with an index `num_assignments` used to specify how many of these references have been assigned correctly. This array should be preallocated to be the size of the maximum possible number of assignments per update event.

The `updates` array is preallocated to be the size of the maximum possible number of update events for any state. An integer, `num_updates` is stored to represent how many of the update events actually contain relevent data at any one time.

Combined algorithm for all Model Types

It was decided that rather than design different event set calculation algorithms for each type of PRISM model, it would be better to analyse the similarities and differences between the three models. This was so generic algorithms could be designed so that they could be overridden by subclasses for each type of model, which could deal with the differences.

The solution is in fact the same for CTMCs and MDPs, with only a small change required to DTMCs to account for the normalisation of probabilities. The analysis of this problem was done in Section 3.3. For each type of model, it is necessary to calculate a set of commands that evaluate to true over s.

The algorithm manipulates a preallocated 3-dimensional array, `true_commands` of references to `Command` objects, which evaluate to true over s. The first dimension of the array indexes the module index of the true command, and the second dimension indexes the action-label of the command. The size of the first dimension is equal to the number of modules in the model + 1. This is so the index [`num_modules`] can be used for synchronisation purposes. The size of the second dimension is equal to the number of action labels + 1. This is so the index [`num_action_labels`] can be used for commands which have no action-label.

For MDPs and CTMCs it is possible to have more than one command of the same action-label in the same module evaluate to true for the same state. Therefore, the third dimension is used to index this. In order to determine how many commands are true in each slot of `true_commands`, a 2-dimension array, `num_true_commands` is stored to count this.

The pseudocode for the first stage of population of `true_commands` is given below. The input parameter is an array, `vars` of integer which is the same size as `model_variables`.

```
Populate_True_Commands(Command[][][] true_commands, int[][] num_true_commands)
 1.  Set each element in num_true_commands := 0
 2.  for each Command c in command_table
 3.      if c.guard.Evaluate(vars)  then
 4.          if c.action_index == -1 then
 5.              true_commands[c.module_index]
 6.                  [num_action_labels]
 7.                  [num_true_commands[c.module_index][num_action_labels]++]
 8.                      = c
 9.          else
10.              true_commands[c.module_index]
11.                  [c.action_index]
12.                  [num_true_commands[c.module_index][c.action_index]++]
13.                      = c
```

In order to deal with synchronisation, it is first necessary to sort out which action labels which are *blocked* over s. Essentially, if for the ith module index and the jth action label index, `alphabet[i][j]` is true, if `num_true_commands[i][j]` is 0, then the jth action label is blocked. In which case all entries in `true_commands` where the the second index is `j` are no longer needed and so `num_true_commands` for that index is set to 0. This is summarised in the following pseudocode:

```
Sort_Out_Blocking(Command[][][] true_commands, int[][] num_true_commands,
bool[][] alphabet)
1.  i := 0
2.  for i < num_action_labels increment i
3.      j := 0
4.      for j < num_modules increment j
5.          if alphabet[j][i] then
6.              if num_true_command[j][i] = 0 then
7.                  k := 0
8.                      for k < num_modules increment k
9.                          num_true_commands[k][i] := 0
```

`true_commands` now effectively contains the set, $synch_a$ for each action label a, as described in Section 3.3. The next stage is to create a `Command` object c_{synch_a} which is the product of the commands for each action label in `true_commands`. This is acheived by the following pseudocode:

```
Form_Products(Command[][][] true_commands, int[][] num_true_commands)
 1.  i := 0
 2.  for i < num_action_labels increment i
 3.      j := 0
 4.      for j < num_modules increment j
 5.          if num_true_commands[j][i] == 0 then continue
 6.          else ifnum_true_commands[num_modules][i] == 0 then
 7.              k := 0
 8.              for k < num_true_commands[j][i] increment k
 9.                  true_commands[num_modules][i]
10.                      [num_true_commands[num_modules][i]++] :=
11.                          true_commands[j][i][k]
12.          else
13.              k := 0
14.              for k < num_modules increment k
13.                  l := 0
14.                  for l < num_true_commands[j][i] increment l
15.                      true_commands[num_modules][i]
16.                          [num_true_commands[num_modules][i]++] :=
17.                              Product_Commands(true_commands[num_modules][i][k],
18.                              true_commands[j][i][l])
```

This algorithm puts the resulting commands in the *sorted* slot for each action label e.g. for the jth action index, the sorted slot is `true_commands[num_modules][j]`. All that is required is a function for forming the product of two Command objects. This is shown in the following pseudocode:

```
Command Product_Commands(Command comm1, Command comm2)
 1.  Create a new Command object, prod
 2.  prod.guard := undefined
 3.  prod.action_index := comm1.action_index
 4.  Create an array, merged of Update objects of size comm1.num_updates*comm2.num_updates
 5.  k := 0
 6.  i := 0
 7.  for i < comm1.num_updates increment i
 8.      j := 0
 9.      for j < comm2.num_updates increment j
10.          Create a new Update object, u
11.          u.prob := a new RealTimes object with comm1.updates[i].prob and
12.              comm2.updates[j].prob
12.          Add each Assignment in comm1.updates[i].assignments to u.assignments
13.          Add each Assignment in comm2.updates[j].assignments to u.assignments
14.          merged[k++] := u
15.  prod.updates = merged
16.  return prod
```

`true_commands` now contains all of the data required to populate the updates event set, `updates`. This is done by extracting the updates for each remaining `Command` in the `true_commmands`. For the ith command, `c`, for each update `u` in `c.updates` modify `updates[i]` such that:

- `update[i].prob := c.prob.EvaluateDouble(vars);`
- `update[i].dist_index := i;`
- `update[i].module_index := c.module_index;`
- `update[i].action_index := c.action_index;`
- `update[i].assignments := u.assignments`.

For discrete-time Markov chain models, each update in `updates` must be normalised. If the total number of remaining commands in `true_commands` is n, for each `UpdateEvent`, u in `updates`:

- `u.dist_index := 0;`
- `u.prob := u.prob / n`.

5.2.4 Making Manual Choices

For a current, s, the previous section described how to generate the possible set, E_s of update events to move into a state s'. This is stored in an array, `updates`, of `Update_Event` objects.

Suppose s is the current state and the ith update is selected from `updates` calculated for s. The simulator engine has to apply the assignments in `updates[i].assignments` to update `model_variables` and then update `the_path`.

For CTMCs, the simulator engine has to also calculate the time spent in the last state, if this was not specified by the user. This calculation requires the generation of a random real number, *sample*, between 0 and 1. With this number the time spent in the last state is calculated as: `(-1*log(sample))/updates[i].prob`

This is shown in the following pseudocode:

```
Make_Choice(int i, real time_in_last_state)
1.  for each Assignment a in updates[i].assignments
2.      Create a temporary integer a_temp
3.      a_temp := a.assign.Evaluate(model_variables)
4.  for each Assignment a in updates[i].assignments
5.      model_variables[a.index] := a_temp
6.  the_path.Add_State(time_in_last_state, i, updates[i].action_index)
```

5.2.5 Making Automatic Choices

The process of making an automatic choice from an update set, E_s, where s is the current state is just a simple case of sampling from the probability distribution in the `updates` array.

If an array of reals, `distribution` is created to be the same size of `updates` where `distribution[i] := updates[i].prob`, the following recursive algorithm can be applied which returns a sample index, where the probability of selecting the ith element is `distribution[i]`. It assumes there is a function `Random_Uniform` which returns a random real number between 0 and 1.

```
int Sel_From_Dist(real[] distribution)
1.   return Sel_From_Dist_Helper(0, 1.0, distribution)
```

```
int Sel_From_Dist_Helper(int start, double max_prob, real[] distribution)
1.  if start ⩾ distribution.length
2.      return start-1
3.  else if Random_Uniform()*max_prob < distribution[start]
4.      return start
5.  else
6.      return Sel_From_Dist_Helper
7.          (start+1, max_prob-distribution[start], distribution)
```

The simple case - CTMCs

For continuous-time Markov chains, the process of making an automatic choice starts by calculating `updates` as described previously. Recalling that the probability of a particular event in a CTMC is the rate of that event divided by the sum of the rates of all events in E_s, the first step is to calculate a real number `sum_rates`, which is calculated as the sum of all `event.prob` for each `event` in `updates`.

An array, `dist` is then populated with the rates from each each element in `updates`. The automatic choice is done by calculating `Select_From_Distribution_Helper(0, sum_rates, dist)` to give an integer, `sample`.

Finally, `model_variables` and `the_path` are updated by calling:

```
Make_Choice(sample, (-1*log(Random_Uniform()))/updates[sample].prob)
```

Optimisations for DTMCs and MDPs

For discrete-time Markov chains and Markov decision processes, it is possible to perform automatic updates without actually calculating the entire `updates` array. It should be noted that this is not possible for continuous-time Markov chains, because probabilities are calculated from the sum of the rates of each element in `updates`. The idea behind this optimisation is given below:

- `Populate_True_Commands` and `Sort_Out_Blocking` still need to be executed.
- It is possible to determine the number of asynchronous commands, `count_asynch`, and how many synchronised commands there will be after all products have been formed, `count_synch`;
- Select a random number $0 \leqslant$ `sample` $<$ `count_asynch` + `count_synch`;
- If `sample` $<$ `count_asynch`, then there is no need to deal with any synchronisation. Suppose `comm` is the `sample`th `Command` object in `true_commands` An array `dist` is then populated with the probabilities calculated for each `Update` object, `u`, in `comm` as:

`u.Evaluate_Double(model_variables).`

It is now possible to use `Select_From_Distribution(dist)` to give an index, `i`. The following pseudocode should then be performed (this is similar to `Make_Choice` except does not require the `updates` array.

```
1.  for each Assignment a in comm.updates[i].assignments
2.      Create a temporary integer a_temp
3.      a_temp := a.assign.Evaluate(model_variables)
4.  for each Assignment a in comm.updates[i].assignments
5.      model_variables[a.index] := a_temp
6.  the_path.Add_State(time_in_last_state, -1, comm.updates[i].action_index)
```

- If `sample` $\geqslant$ `count_asynch`, then a synchronous command has been selected. In this case, for the `sample`th action index, `index`, products have to be formed of all commands. The resulting command can be selected from in the same way as `comm` was handled in the last step.

The optimisation is based upon the assumption that for both types of model, the choice between probability distributions is always a uniform probabilistic choice[3].

It should also be noted that it is not possible to calculate the `choice_made` parameter for the `Add_State` function without calculating `updates`. Therefore, if this information is required, it is necessary to calculate `updates`.

This optimisation was developed because this operation is perhaps the most frequently occurring in the approximate model checking algorithms.

5.2.6 Loop Detection

In order to design the algorithms for the detection of loops in `the_path`, the following aspects were considered:

- An algorithm which analyses `the_path` and can return an answer as to whether it is looping would be computationally expensive, as for each addition, the algorithm would have to analyse the entire path. This approach does have the benefit that when the path is modified, i.e. for backtracking, loop detection is easy to re-calculate.
- An approach which detects loops as they happen is more difficult to design, but results in a more efficient approach. In this case, only a small subset of the path has to be analysed for each addition.

For efficiency reasons, the second approach was chosen. Therefore, for each addition to the path, i.e. every time `the_path.Add_State` is called, `the_path` has to be queried for loops.

A loop can only occur in a portion of the_path that is *looping deterministically*. That is, for every state, s, in the looping portion, the update event set, `updates`, can only contain one possible destination for each `UpdateEvent`, `u` when `u.assignments` are applied to s. Furthermore, the state at `the_path[current_index].variables` should be the same as the the `variables` of another state in the looping portion.

The algorithm works by maintaining an index, `potential_loop_start` to the start of deterministic portions of `the_path`. Initially, `potential_loop_start` is set to -1 to indicate that there is no deterministic portion. While `potential_loop_start` is still -1, if the update event set, `updates` for a particular state is deterministic, then `potential_loop_start` is set to be the index of that state in `the_path`. While `potential_loop_start is >= 0`, every time a new state is added to `the_path` and its update event set is calculated, `the_path` is searched from `potential_loop_start` to `current_index-1`. If a state at index, `start`, is found that is the same as `model_variables` then a loop has been detected. A Boolean variable, `loop_detected` (which is initially false for each path) is set to true and an integer variable, `loop_start` is set to `start`.

[3] This was the assumption made in the Background section for MDPs as the way to resolve the non-determinism.

5.2.7 Evaluation of State Label Formulae

One of the requirements of the simulator user interface is that it must have a list of state label formulae. It was decided that this would be stored as part of the simulator engine, as an array `loaded_state_formulae` of `NormalExpression` objects. This array can be populated whenever a label formula is added to the simulator user interface.

Important Functions

A few functions are provided which query the formulae stored in `loaded_state_formulae`:

- A function, `Query_Formula(int i)` is provided which returns

  `loaded_state_formulae[i].Evaluate(model_variables)`,

  i.e. whether the ith loaded formula is satisfied in the current state.

- A function, `Query_Formula(int i, int step)` is provided which returns

  `loaded_state_formulae[i].Evaluate(the_path[step].variables)`,

  i.e. whether the ith loaded formula is satisfied in state in `the_path` indexed by `step`.

Furthermore, functions are required to query whether the current state, or a state in `the_path` is the initial state, or a deadlock state:

- A function, `Query_Initial()` is provided which returns whether `model_variables` is the same as `the_path[0].variables`, i.e. whether the current state is the initial state;
- A function, `Query_Initial(int step)` is provided which returns whether `the_path[step].variables` is the same as `the_path[0].variables`, i.e. whether the state in`the_path` indexed by `step` is equal to the initial state;
- A function, `Query_Deadlock()` is provided which returns whether the update event set for `model_variables` is empty, i.e. whether there are no outgoing transitions from the current state;
- A function, `Query_Deadlock(int step)` is provided which returns whether the update event set calculated for `the_path[step].variables` is empty.

5.2.8 Evaluation of Path and Reward Formulae

Path and reward formulae such as `[true U<=3 s=3]` or `[F s=3]` are required for two purposes:

- One of the requirements of the simulator user interface is that it must have a list of path and reward formulae;
- For use in statistical sampling techniques.

Therefore is was decided that these formulae would be stored as part of the simulator engine, as an array: `loaded_path_formulae` of `PathFormula` objects. Each object maintains a boolean `answer_known` as to whether the result for particular path or reward formula is known according to the current state of `the_path`. If the result is known, then a boolean `answer` is stored which contains the answer.

In order to determine this, each `PathFormula` object must be notified whenever the `the_path.Add_State()` function is called. When this happens the `PathFormula` may update its `answer` and `answer` known variables appropriately. This is done in an abstract method, `Notify_Current_State()` which should be implemented in subclasses for each different type of formula.

Each `PathFormula` also provides a function, `Is_Answer_Known()` which by default returns true if `answer_known` is true or if `proven_looping` is true.

For rewards formulae, objects can be of type `RewardFormula` which *extends* `PathFormula`. The only real customisation is that a double `answer_double` is stored because each path will have a real valued reward, rather than a Boolean value.

With the `PathFormula` and `RewardFormula` classes defined like this, all that is required is to define the `Notify_Current_State()` method for each subclass. These are discussed in the following sections and the inheritance hierarchy is given in Appendix B as Figure B.1.

Bounded Until

A `BoundedUntil` object *extends* `PathFormula` and stores all of the relevant information for a bounded until path formula:

- The left expression, `expr1`, which is an `Expression` object;
- The right expression, `expr2`, which is an `Expression` object;
- The lower bound, `lower_bound`, which is stored as a real number;
- The upper bound, `upper_bound`, which is stored as a real number;

Furthermore, for each execution path, a `BoundedUntil` object maintains a integer, `counter` which records the number of steps so far, and a real number, `time_so_far` which records the elapsed time so far. `counter` is initially -1 for every path.

The function `Notify_Current_State()` essentially checks whether if the time so far is within in bound if the right expression evaluates to false then the left must still be true. If the left expression is true and the right expression is false then the `answer = false`. If the right expression evaluates to true in the time bound then `answer = true`.

Until

An `Until` object *extends* `PathFormula` and stores all of the relevant information for an unbounded until path formula:

- The left expression, `expr1`, which is an `Expression` object;
- The right expression, `expr2`, which is an `Expression` object;

The pseudo-code for the overriding of `Notify_Current_State()` is given below: `counter` is initially -1 for every path.

```
Notify_Current_State() for Until
1.  if !answer_known then
2.      if expr2.Evaluate(model_variables) then
3.          answer_known := true
4.          answer := true
5.      else if expr2.Evaluate(model_variables) then
6.          answer_known := true
7.          answer := false
```

Next

A `Next` object *extends* `PathFormula` and stores all of the relevant information for a "next" path formula:

- An expression, `expr`, which is an `Expression` object;

Furthermore, for each execution path, a `Next` object maintains a integer, `counter` which records the number of steps so far. `counter` is initially -1 for every path.

The pseudo-code for the overriding of `Notify_Current_State()` is given below:

```
Notify_Current_State() for Next
1.  counter++
2.  if !answer_known then
3.      if counter = 1 then
4.          answer_known := true
5.          if expr.Evaluate(model_variables) then
6.              answer := true
7.          else
8.              answer := true
```

Cumulative Rewards

A `CumulativeReward` object *extends* `RewardFormula` and stores all of the relevant information for a cumulative reward formula:

- The time bound, `time_bound`, which is stored as a real number;

Furthermore, for each execution path, a `CumulativeReward` object maintains an integer, `counter` which records the number of steps so far, and a real number, `time_so_far` which records the elapsed time so far.

The function `Notify_Current_State()` works by checking whether the time so far is greater than the time bound. If so, `answer_double` is set to be the `the_path[current_state-1].path_reward`.

Instantaneous Rewards

An `InstantaneousReward` object *extends* `RewardFormula` and stores all of the relevant information for an instataneous reward formula:

- The time bound, `time_bound`, which is stored as a real number;

Furthermore, for each execution path, a `CumulativeReward` object maintains an integer, `counter` which records the number of steps so far, and a real number, `time_so_far` which records the elapsed time so far.

The function `Notify_Current_State()` works in the same say as for CumulativeReward except `answer_double` is set to `the_path[current_state-1].state_reward`.

Reachability Rewards

A `ReachabilityReward` object *extends* `RewardFormula` and stores all of the relevant information for a reachability reward formula:

- An expression, `expr`, which is an `Expression` object;

The pseudo-code for the overriding of `Notify_Current_State()` is given below:

```
Notify_Current_State() for Until
1.  if !answer_known then
2.      if expr.Evaluate(model_variables) then
3.          answer_known := true
4.          answer_double := the_path[current_index].path_cost
```

5.2.9 Approximate Model Checking Algorithm

The approximate model checking algorithm was extended in Figurefig:gaa-multiple in order to deal with multiple properties. This section considers the design of this algorithm as part of the simulator engine.

Properties for approximate model checking are stored in an array, `properties` of `Sample_Holder` objects.

The Sample Holder

A `Sample_Holder` object, `sh` stores all of the data required to perform approximate model checking on a particular path or reward formula stored as a PathFormula object `sh.formula`:

- A real value, `cumulative_value`, which for path formula is the sums of 0s and 1s and for rewards formula is the sum of rewards.
- The number of samples so far, `no_samples`.

The following functions are provided:

- `Sample(real sample)` - This increments `no_samples` and adds `sample` to `cumulative_value`.
- `bool Done()` - This returns true if number of samples is greater than the number of iterations, N.
- `Get_Result()` - This returns `cumulative_value` / `no_samples`, which is the result.

Implementation of the Extended Generic Approximation Algorithm for Multiple Properties

The following pseudocode performs the approximate model checking technique for N iterations:

```
GAA_extended(int N)
1.  initial := clone of initial state
2.  iter := 0
3.  for iter < N incrementiter
4.      model_variables := initial
5.      current_index := -1
6.      Add_State(current_index, -1, -1)
7.      Make automatic choices until all formulae have answer_known = true
7.      or until k is reached
8.      for each Sample_Holder sc in properties
9.          sc.Sample(sc.formula.Get_Answer_Double())
10.         textrmresetsc.formula
```

5.3 Simulator User Interface Design

The simulator user interface is built on top of the simulator engine and uses its functionality as described in the specification. It is also built on top of the existing PRISM user interface. When considering the design of the simulator user interface, the following factors were taken into account:

- All components included in the specification needed to be laid out in an intuitive way;
- An existing layout had already been developed for the PRISM simulator prototype, which was reasonably successful;
- The specification requests the view of the current execution path be customisable.

5.3.1 Component Layout

It was decided that components would be grouped according to functionality. For example, all buttons that concern the current execution path are grouped together and all components that concern the current update event set are grouped together.

The process of creating the user interface was iterative, with user input taken into account after each iteration. The chosen layout is shown in Figure 5.2.

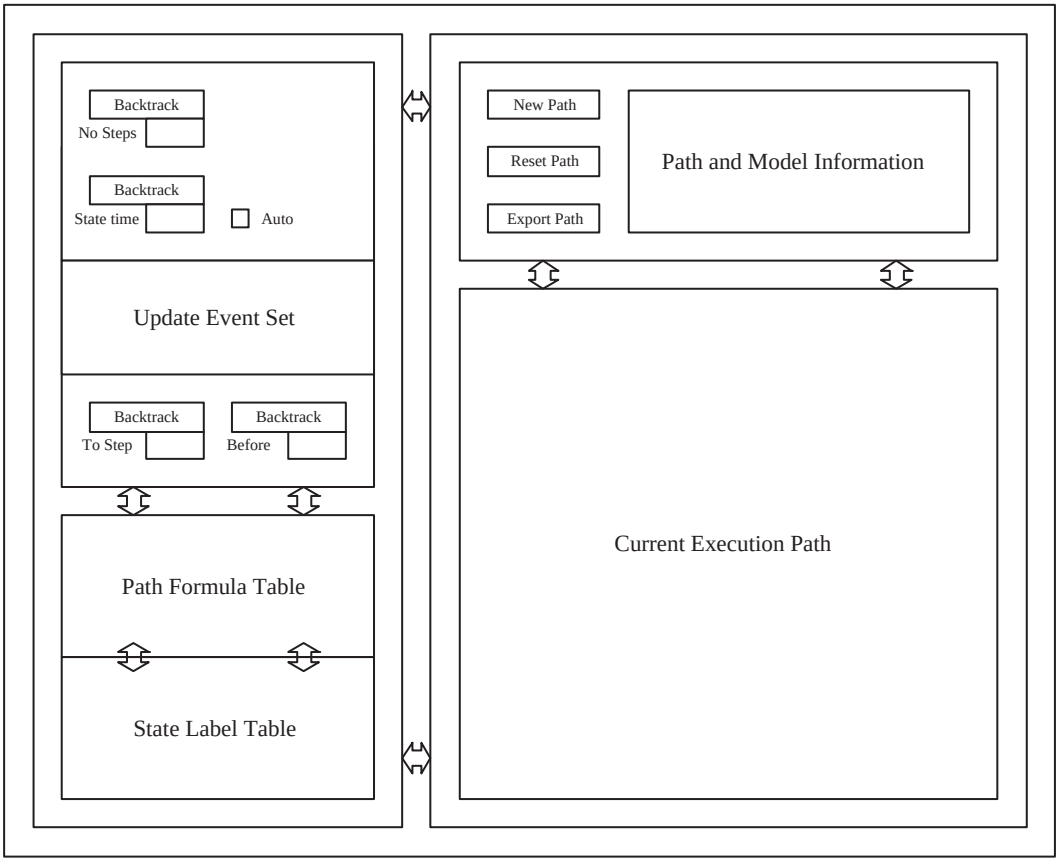


Figure 5.2: Layout of the simulator user interface

5.3.2 Current Execution Path Table

The specification suggests the current execution path table should display a grid of data values, with columns for the behaviour of each variable over time. However, for most models, only a small subset of variables actually change from one state to another. It was decided that rather than display the grid of data values,only changes in values should be rendered to create a *sequence diagram* style for the execution path. However, when the user selects a particular state, the values for that row are re-displayed. A portion of a sample path rendered using this design is shown in Figure 5.3.

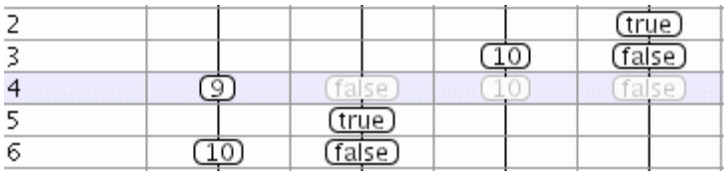


Figure 5.3: A portion of a path rendered using the sequence diagram style

5.4 Implementation Issues

The PRISM graphical user interface is written in the Java programming language. Most of the computational engines of PRISM are implemented using C and C++, using the Java Native Interface to communicate with Java.

Because one of the key parts of the project was that the simulator should be as fast as possible, it was decided that the simulator engine would also be written in C++. The hope was

that a compiled language such as C++ would perform faster than the interpreted Java.

One important issue to note is that PRISM has to be released on three different platforms: Windows, Linux and Solaris. For Java, this is not an issue, but all C++ code had to conform to ANSI C++ standards, so that it was compiler independent.

5.4.1 The Simulator Engine

The interface for the simulator engine was written in the Java, using native methods to integrate with the C++ code. All of the model, path, expression and path formula data structures were implemented in C++.

The decision to use C++ for the simulator engine did however cause many problems throughout the project. Most problems were to do with memory management and memory leaks, issues that do not arise in Java programming.

5.4.2 Simulator User Interface

PRISM allows new graphical user interface components to be added via its "plug-in" mechanism. This mechanism provides an abstract class `GUIPlugin`, which provides all base functionality of integrating with the PRISM graphical user interface:

- It allows new plug-ins to communicate with existing parts of the system via a message passing protocol. For the simulator user interface, this was used to listen for when new models were loaded and parsed into PRISM.
- It allows the plug-in to be displayed within the PRISM GUI as a new tab panel, with menus and options panels. The options panel feature was utilised to integrate a simulator options panel.

The simulator user interface itself extended `GUIPlugin` and was constructed using Java Swing components. The layout of the interface was put together using the Netbeans form editor tool. For components such as the execution path table, it was required to use inheritance to override the way that Swing renders its components. This technique was used to render the sequence diagram style and for the display of loops.

5.4.3 Properties Integration

The integration of the statistical sampling techniques into the PRISM graphical user interface involved the modification of several PRISM graphical user interface components. These components included:

- The properties list had to have a new "Simulate" button integrated into its popup menu;
- The experiments dialog had to have a new "Use Simulation" check box added to it;
- The mechanism for handling PRISM experiments had to be considerably modified to cope with using results from the simulator engine rather than from PRISM's numerical engines.

Despite having to understand and manipulate a lot of existing components, the integration of the statistical sampling techniques was relatively problem free.

5.4.4 Implementation Tools

For code written in C++, Microsoft's Visual Studio .NET 2003 was used. This was particularly useful because it enabled the handling of the large amount of source files involved in the PRISM tool.

For Java editing, the tool Netbeans was used. This tool also provided a useful mechanism of managing large numbers of files. The most useful feature of this tool was its form editing utility, which allows powerful user interfaces to be developed very quickly.

Chapter 6

Testing

This section discusses the techniques that were used test the PRISM simulator at various stages of the software lifecycle. Section 6.1 describes how the requirements were validated by the development of prototypes. Section 6.2 states the strategies employed to rigourously test the most important units of the tool.

Once developed, the PRISM tool was applied to a wide variety of case studies to test the performance of the system and the correctness of the statistical sampling techniques. This is discussed in Section 6.3.

The tool was then released to a small number of users as an alpha release. This is discussed in Section 6.4.

6.1 Requirements Validation

The process of validating requirements started with the development of the PRISM simulator prototype. Because of this tool, it was possible to see which parts of the specification were useful and which parts required further development.

The requirements of the prototype that were successfully validated were incorporated as part of the requirements for this project. In order to continue this process, an expert user was asked to analyse any new requirements and state whether they were correct. A number of proposed features that would not have provided any real benefit were eliminated in this way.

6.2 Unit Testing

The development of the most important algorithms of the system included rigourous unit testing techniques. For example, the generation of the update set for a particular state is one of the most frequently executed functions in the simulator. In order to test its correctness, a test was performed that for a wide selection of PRISM models, the transition matrix generated by PRISM was compared state by state to the update set calculated by the simulator. In this case, a number of errors were found and corrected.

Most tests required results calculated by PRISM to be used as a reference.

6.3 System Testing

When most of the components of the PRISM simulator had been developed, the whole system was tested. The following elements were investigated:

- The scalability of the simulator to very large PRISM models;
- The robustness of the simulator user interface,
- The correctness of the statistical techniques, as compared with results calculated by PRISM.

6.4 Alpha Testing

In order to test the usability of the tool, the PRISM simulator was introduced to a wide variety of different users, including:

- The PRISM development team;
- A group of students using PRISM for the first time for a coursework exercise.

The alpha testing process also provided further means of testing the aspects investigated in system testing.

Chapter 7

Project Management

This chapter describes how the PRISM simulator project was managed. Section 7.1 describes how the development lifecycle was planned and Section 7.2 indicates how the actual development of the project differed from the original schedule. Finally, an appraisal of the project management methodology is given, describing the lessons learnt.

7.1 Work Breakdown

The development life cycle used in this for this project was iterative. The result of the initial planning around this life cycle was a work breakdown structure, which is summarised below:

- Project Planning and Initialisation - All work concerning planning, proposal writing and project standards should be done in the first phase.
- Requirements and Graphical User Interface Design - All of the explicit requirements of the system should be collected in the second phase. The user interface should be designed at the same time as the elicitation process to ensure that the specification is as complete and accurate as possible.
  - Domain Analysis - Although some domain analysis will have already been performed, throughout the elicitation and analysis process, it will be important to gain as much necessary knowledge as possible.
  - Planning - Strategies and techniques should be found for eliciting requirements. For each technique selected, all of the appropriate preparations should be made including arranging meetings with users and developers.
  - Elicitation - The strategies outlined in the requirements elicitation plan should be applied. This stage will involve conducting meetings with potential users of the system. These meetings should be used to draw up and agree upon a set of user requirements.
  - Analysis - A requirements document should be produced and approved by potential users of the system.
  - Design for GUI - At the same time as requirements elicitation, a design for the user interface should be produced. Suggestions should be drafted as simple prototypes for/in meetings (possibly using tools such as MS-Visio or Netbeans) and then modified according to any feedback from users.
- 1st Iteration: Core Data Structures and Algorithms - The foundations of this project involve the design and implemention of core data structures and algorithms that will be used in future iterations. The majority of the work will be performed in this phase, the final result being a simple command line simulator, which tests all of the necessary functionality.
- 2nd Iteration: Simulator User Interface - This phase involves research into the existing PRISM GUI, and then the production an efficient class design for the new user interface. The user interface should then be implemented and tested. It will be important to consider that for the second project extension, it may be necessary to add to this user interface in order to incorporate graphical module diagrams.

- 3rd Iteration: Properties User Interface Integration - The PRISM GUI includes an interface that deals with all model checking operations. This includes the verification of properties against a model using one of PRISMs model checking engines. In this phase the simulator's data structures and algorithms from should be integrated into the existing properties user interface .

The following phases were considered as possible extensions to the system, listed in order of priority.

- 1st Extension: Rewards Integration - A current development of the PRISM tool allows paths to be given a reward value. This phase invovles the integratation of the idea of rewards into the manual and automatic exploration algorithms and to allow reward based properties to be verified using the statistical sampling techniques.
- 2nd Extension: Graphical module integration - A current development of the PRISM user interface is the Graphical Model Editor. This allows PRISM models to be specified diagrammatically. This extension would enable users of the simulator user interface to see the current state of graphical modules diagrammatically as they step through the model.
- 3rd Extension: MDP Automatic Path Generation: Further Research and Improvements - The initial work into MDP simulation will only be very approximate. This extension is to perform further research into the area, with the hope of producing a more effective solution.

The detailed breakdown of each of these phases is given in the *Project Plan*, which is given as a supporting document to this dissertation.

7.2 Scheduling

This section compares the original project schedule with the actual series of events that led to the project's completion.

Phase	Original schedule	Actual times	Explanation
Project Planning and Initialisation	July to Mid September 2004	July to Mid September 2004	
Requirements and GUI Design	Mid September to Mid October 2004	Mid September 2004 to Mid November 2004	Domain research and analysis was much more complicated than anticipated. Also, the process of documenting requirements took longer than expected.
1st Iteration	Mid October to Mid November 2004	Mid November 2004 to Mid February 2005	This work was considerably more complex than the original estimates. However, this time was also spent developing the 1st extension of the project because it was decided that concurrent development would be more appropriate. Also, much of January was lost due to unforseen non-project related circumstances.

2nd Iteration	Mid November 2004 to January 2005	March to Mid March 2005	This work was actually performed last because it was decided that the "Properties User Interface" should be given priority. However, with the ability to do 60 hours per week, it was possible to complete this iteration before the presentation deadline.
3rd Iteration	January to Mid February 2005	Mid February to March 2005	Due to prior experience of the properties user interface, this work was performed much more quickly and problem free than expected.
1st Extension	Mid February to March 2005	January to Mid February 2005	It was decided that the priority of the rewards extension was actually higher than the user interface iterations and so it was developed along with the core data structures and algorithms.

Unfortunately, time did not allow the 2nd and 3rd extensions to be developed.

7.3 Appraisal

The original work breakdown and project schedule provided a very useful means of partitioning the project into manageable units and of monitoring progress via the various milestones. For example, in January, it seemed unlikely that some of the user interface work would be completed, because many deadlines were still overdue. This allowed priorities to be reassessed and resulted in a re-scheduling of the rewards extension and the properties user interface.

The iterative lifecycle proved successful for this type of project, although in a lot of cases, it was not as rigidly applied as the work breakdown structure would suggest. This was mainly due to the fact that it was necessary to design and implement parts of the software in order to understand some of the background material or to understand or validate the requirements.

The main lesson learnt from this project was that it is important to consider the risk of unforseen consequences by providing several fallback positions should the full range of functionality prove unfeasible.

Chapter 8

Evaluation

This chapter gives an evaluation of the PRISM simulator. The simulator user interface is appraised in terms of its usability, scalability, robustness, responsiveness and specification conformance. The following sections give two case studies that attempt to show the success of the approximate verification techniques by comparing results and performance to PRISM, as well as to similar techniques of related tools.

8.1 Simulator User Interface

The main motivation for the simulator user interface was to give users the opportunity to further understand the models they develop. In order to evaluate the success of the simulator user interface, a number of users were given the opportunity to use it as part of their model development process. As well as providing a useful means of obtaining further user requirements, these tests allowed feedback to be gathered.

It should be noted that the software released was a development version of the PRISM simulator. Any errors found by users were immediately corrected. The feedback is summarised below:

8.1.1 Usability

The PRISM simulator has already been used extensively by members of the PRISM development team to debug their PRISM models and to help users from other universities with their problems. The team have found that the tool makes light work of spotting modelling errors that otherwise would have had to have been searched for manually.

One particular student used the tool whilst developing a PRISM model from scratch for a coursework assessment within the university. One result that was not anticipated was that simulator was a helpful teaching tool of probabilistic modelling and the PRISM language. The student stated that he was able to understand the more advanced features of the PRISM language, such as module synchronisation and label formulae, more quickly because of the simulator. As well as this, a number of modelling errors were found, that had not been noticed during modelling or in analysis. When these errors were resolved, verification results were significantly different and were closer to the expected results.

Many users commented on how some of the visual features were successful:

- When asked about usefulness of the *sequence diagram* style for the execution path table as compared to the ordinary *plain table* style (these are illustrated in Figure 8.1), users commented on how it enabled them to see patterns and therefore better understand the execution behaviour of their models.

- The feature which allows users to select a particular *label*, and then highlights all of the states in the current execution path which satisfy the predicate for that label, was liked by users. Again, this was because it made it easier to spot patterns. For example, Figure 8.2 gives an example where the selected label formula is very rarely false and is relatively complex. Therefore, the non-satisfying states would be very difficult to spot without this feature.

Sequence Diagram Style

Step	left_n	left	right_n	right	r
0	10	false	10	false	false
1			9		
2				true	true
3			10	false	false
4	9	false	10	false	false
5		true			true
6	10	false			false
7			9		
8				true	true
9			10	false	false
10			9		
11				true	true
12			10	false	false
13	9				
14		true			true
15	10	false			false
16			9		
17				true	true
18			10	false	false
19	9				
20	9	true	10	false	true

Plain Table Style

Step	left_n	left	right_n	right	r
0	10	false	10	false	false
1	10	false	9	false	false
2	10	false	9	true	true
3	10	false	10	false	false
4	9	false	10	false	false
5	9	true	10	false	true
6	10	false	10	false	false
7	10	false	9	false	false
8	10	false	9	true	true
9	10	false	10	false	false
10	10	false	9	false	false
11	10	false	9	true	true
12	10	false	10	false	false
13	9	false	10	false	false
14	9	true	10	false	true
15	10	false	10	false	false
16	10	false	9	false	false
17	10	false	9	true	true
18	10	false	10	false	false
19	9	false	10	false	false
20	9	true	10	false	true

Figure 8.1: Comparison of the different styles of path table.

Formula label highlighted:

```
"premium" = (left_n>=left_mx&Toleft_n)|
(right_n>=right_mx&Toright_n)|
((left_n+right_n)>=left_mx&Toleft_n&line_n&Toright_n)
```

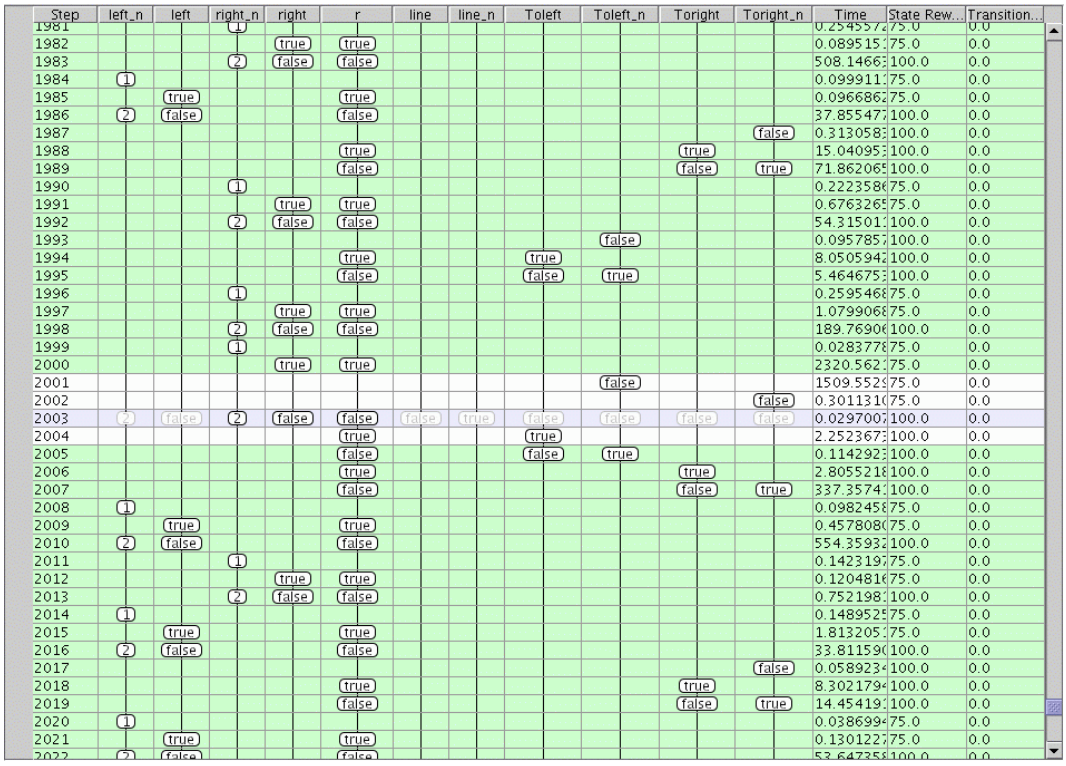


Figure 8.2: Highlighting the satisfaction of state formula *labels*

- Users who developed models with a large number of variables found that the column hiding feature was useful. In particular, they liked how it was possible to hide multiple columns in one selection of the menu.

8.1.2 Scalability

A member of the PRISM development team wanted to use the simulator for a very large model. This model had 523 variables and over 7000 commands. Originally, the simulator user interface did not scale up to such a large model, but after a few modifications, it worked correctly and reasonably quickly: To calculate a path of 100 steps takes around 3 seconds. For each step, the simulator hass to evaluate over 7000 guards and then sample from around 100 possible updates.

The only real weakness was that for such a large number of variables, the column hide/show menu could not display all of the variables, even when the application was run with a screen size of 1280 by 1024 pixels.

8.1.3 Robustness

Many features were provided to stop unexpected behaviour, such as validation checks on all text box entries and an error handling mechanism for when parsed PRISM models are loaded. Furthermore, the system has been tested successfully, with a range of valid and invalid data values, on a wide and varied range of existing case studies.

Despite this, there are still a few outstanding issues concerning memory allocation, which in rare cases cause the program to crash.

8.1.4 Responsiveness

For the majority of PRISM models, the user interface works quickly, with all operations being performed almost instantaneously. However, the following exceptions have been observed:

- When a label formula is selected, the highlighting of large paths is often quite slow;
- When the execution path is large, scrolling and menu selection lag behind mouse input;
- When a large path length is set, (e.g. $> 10^6$), the "New Path" function takes a considerable amount of time.

8.1.5 Specification Conformance

All functional requirements for the simulator user interface, specified in Section 4.2, were met by the implemented solution.

8.2 Approximate Model Checking

The approximate model checking techniques developed as part of the PRISM simulator were implemented to provide an alternative to numerical model checking in circumstances where model size and computation times are unfeasibly large. This section gives two case studies designed to give a comparison of the PRISM simulator's model checking results with the results produced by PRISM's numerical engines. Furthermore, the first case study compares results calculated using PRISM simulator with the results calculated using alternative tools and the second demonstrates how the PRISM simulator can be used to check models that are unfeasible to check with PRISM's numerical engines.

8.2.1 Dice Programs

This case study considers a probabilistic program that attempts to model a fair dice using only fair coins. It is taken from the PRISM web site [3].

The Model

The system is modelled as a discrete-time Markov chain. For each state of the model, there are two outgoing transitions, each with a probability of 0.5, modelling the two possible outcomes of tossing a fair coin. When the model enters the terminating state, it assigns a value to the variable `d` which is the value of the dice. Also, for each state, a reward of 1 is accumulated.

The PRISM code for this model is shown in Figure 8.3, and a graphical version has already been shown in Figure 1.2

```
1.  probabilistic
2.
3.  module die
4.
5.      // local state
6.      s : [0..7] init 0;
7.      // value of the dice
8.      d : [0..6] init 0;
9.
10.     [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
11.     [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
12.     [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
13.     [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
14.     [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
15.     [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
16.     [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
17.     [] s=7 -> (s'=7);
18.
19. endmodule
20.
21. rewards
22.
23.    true : 1;
24.
25. endrewards
```

Figure 8.3: PRISM model of a fair dice using only fair coins

Properties

For the purposes of analysis, the following properties were defined:

- "What is the probability of throwing x?" where x is an undefined integer constant:

```
P=?[ true U s=7&s=x ]
```

- "What is the expected reward to reach the terminating state?":

```
R=?[ F s=7 ]
```

Results - PRISM

When the PRISM numerical engines (specifically, the *hybrid* engine) are used to verify these properties, as expected, the probability of throwing any value from 1 to 6 is exactly 1/6 and is 0 for any other value. The expected reward to reach the terminating state is exactly 11/3. Model construction took 0.287 seconds and each property took 0.002 seconds[1] to verify.

Comparison of PRISM Simulator results with PRISM

The PRISM simulator was used to model check the same properties using an approximation parameter $\varepsilon = 0.01$, a confidence parameter $\delta = 10^{-10}$ and a maximum path length $k = 2000$. This means that the the probability of the results being within 0.01 of the actual answer should be 0.9999999999. These parameters mean that 412042 iterations of the approximation algorithm are required.

The results for the probability of throwing x are shown in the following table:

[1] These results and statistics were calculated on a Pentium 4, 2GHz with 1GB RAM running Red Hat Linux.

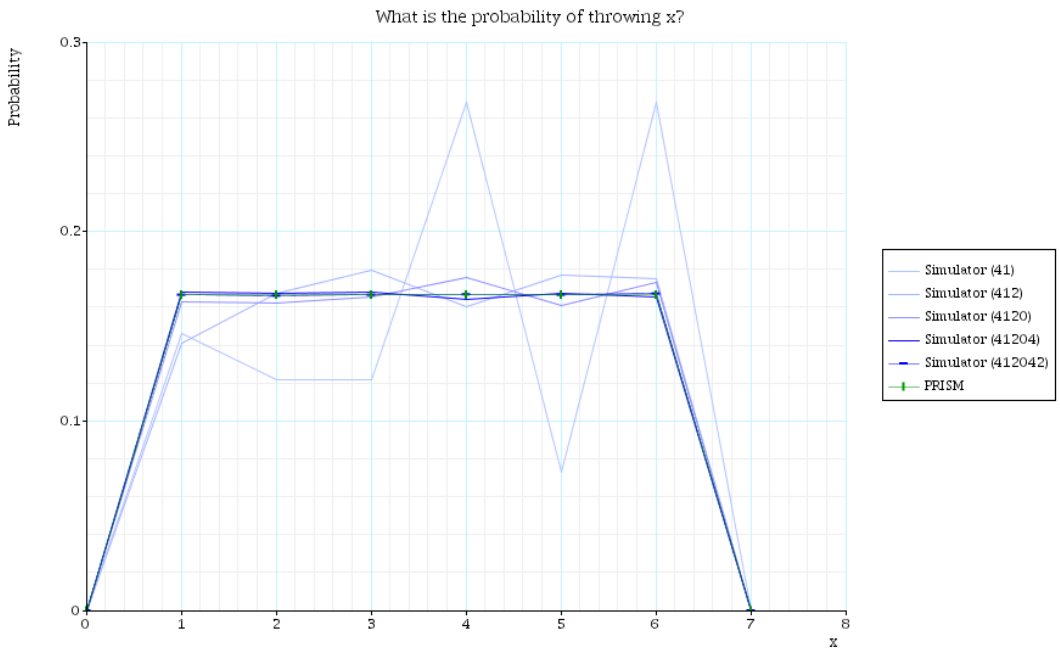


Figure 8.4: Comparison of the PRISM simulator with PRISM for the dice program

	Simulator		PRISM		
x	Result	Time (seconds)	Result	Time (seconds)	Result Difference
0	0.0	3.23	0.0	0.002	0.0
1	0.1666141	3.39	0.1666665	0.002	0.000052
2	0.1661141	3.19	0.1666665	0.002	0.000552
3	0.1666068	3.26	0.1666665	0.002	0.000060
4	0.1668277	3.27	0.1666665	0.002	0.000161
5	0.1665995	3.26	0.1666665	0.002	0.000067
6	0.1672378	3.21	0.1666665	0.002	0.000571
7	0.0	3.51	0.0	0.002	0.0

These results are shown in Figure 8.4, which also shows the effect that changing the number of iterations has on the accuracy of the results.

When the results were calculated *simultaneously*, using the technique outlined in Figure 3.8, the PRISM simulator took 5.21 seconds.

From these results, the following conclusions can be made:

- The difference in results between the simulator and PRISM is never greater than approximation parameter of 0.01. Therefore, it can be concluded the approximation technique is working correctly for this example.
- For this example, PRISM's numerical engines are significantly faster than the PRISM simulator for the purposes of verification.
- When the results are calculated simultaneously the average verification time per property is vastly improved as compared to separate calculation. In this example, the total time was 5 times less. However, greater improvements are possible, as shown in Figure 8.5. This graph demonstrates the clear benefits of the extension to $\mathcal{GAA}$ [6] implemented in the PRISM simulator.

The result for the expected reward to reach the terminating state was calculated as 3.669980 which is very close to the actual answer of 3.666667.

Comparison of PRISM simulator with alternative techniques

The following table compares the verification times of the first property for the PRISM simulator, the prototype simulator and the Approximate Probabilistic Model Checker (APMC), which are

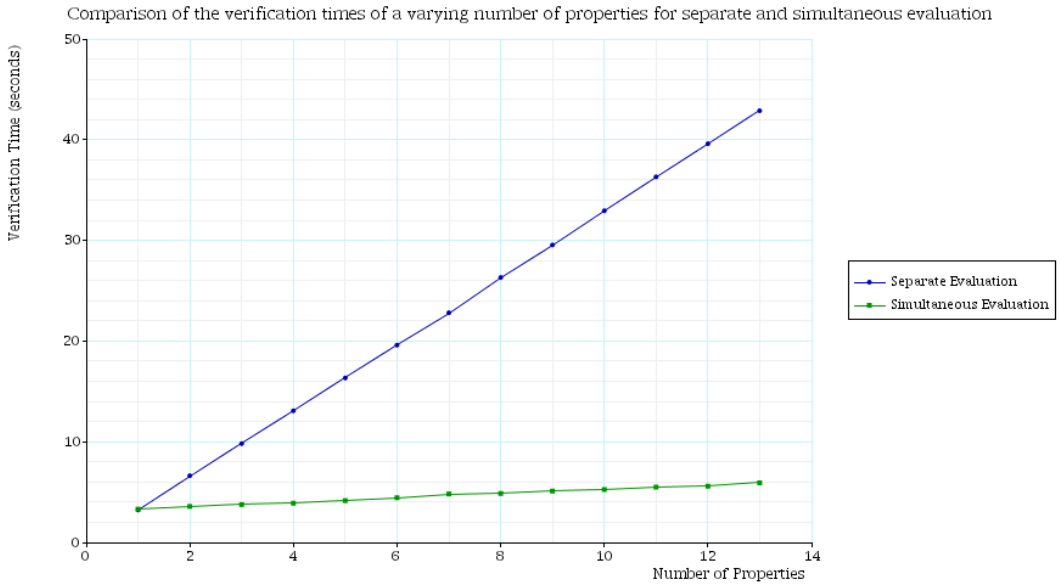


Figure 8.5: Comparison of separate and simultaneous evaluation using the PRISM simulator

three tools that perform approximate model checking. Each property was verified using 412042 iterations and the actual returns were comparable (expect for APMC ($k = 4$)). The numbers for the APMC results represent the chosen value of k.

x	Simulator	Prototype	APMC (4)	APMC (10)	APMC (20)	APMC (200)
0	3.23	22.10	3.77	4.01	6.02	48.19
1	3.39	21.85	1.91	3.83	5.94	47.87
2	3.19	21.99	1.93	3.89	5.90	47.94
3	3.26	21.98	1.93	3.93	5.94	47.95
4	3.27	21.81	1.94	3.91	5.95	47.91
5	3.26	21.82	1.93	3.92	5.95	53.95
6	3.21	21.89	1.93	3.91	5.95	47.95
7	3.51	21.87	3.91	3.96	5.90	47.95
Simul	5.21	30.298	N/A	N/A	N/A	N/A

From these results, the following conclusions can be made:

- Even though in general, the prototype simulator only provides a small subset of the functionality provided by the PRISM simulator, for this simple example the algorithms are the same. i.e. the prototype simulator implements loop detection and the simultaneous verification of properties. The prototype simulator used PRISM's '`ModulesFile`', but it was decided that this structure could be improved for the purposes of simulation, and this was indeed the case. A comparison of the evaluation times for these two tools, shows the clear benefit of and justifies the extra development time of re-designing and implementing the underlying data structures.

- Comparison with APMC is not so straight forward, because APMCs algorithms do not incorporate the extensions added by this project. i.e. loop detection and simultaneous verification of properties. This meant that a direct comparison with identical parameters produced results suggesting that the PRISM simulator was many times faster than APMC. However, these results are misleading: If it is assumed that the average path length to the terminating state is 4:

  - The PRISM simulator will produce paths with an average length of 4, because it will detect the loop in the terminating state.
  - APMC will continue evaluating paths up until length k, whether the loop has been detected or not.

Therefore, if k is set to 4, it is possible to see which tool performs each step of the path generation more quickly. From the table, it can be seen that APMC is around 50% faster when compared in this way. However, in this case the results are always underestimates because many paths are longer than 4. As k gets larger, the results get more accurate and at around $k = 10$ the results are consistently within the error bounds. At this point, verification times are comparable to, if not longer than the PRISM simulator. In conclusion, it can be seen that a clear advantage that the PRISM simulator has over APMC is that a user can just use a value of k that is so large that paths are extremely unlikely to be that long.

- APMC does not provide the feature and therefore enjoy the benefits of simultaneous verification.

8.2.2 Cell Cycle Control in Eukaryotic Cells

This case study investigates a more complex model of a biological system. It is taken from the PRISM web-site [3] and is a translation of a formal specification given in [7].

The cell cycle is a process that occurs in eukaryotic cells, a common class of cells in single or multi-cellular organisms. The process regulates how the cell grows, how DNA is replicated and when mitosis should occur. Each phase of the cell cycle is dependent many complex pathways whose rates depend on a series of *base rates* and the concentrations of different proteins.

The Model

In PRISM, this system can be modelled as a CTMC, with the reaction times being sampled from a negative exponential distribution. The model itself uses many of PRISM's advanced langauge features, such as synchronisation, undefined constants and rewards and is fairly complex: it consists of 7 modules, 13 variables and 37 commands. The PRISM code for this model is given in Appendix C. The table [3] below gives the number of states and transitions for different values of N, a constant which determines the maximum relative amount of each protein in the model.

N	States	Transitions
2	4,666	18,342
3	57,667	305,502
4	431,101	2,742,012
5	2,326,666	16,778,785
6	9,960,861	78,768,799

The purpose of showing this table is to illustrate that when $N > 6$, the number of states increases beyond 10^6 states, which is regarded as the maximum number of states that can be handled by PRISM on a standard desktop PC.

Properties

The property investigated here concerns the amount of cyclin, an important protein in the cell cycle, that is bound to another protein called MPF (mitosis promoting factor). This binding is important, because it promotes the activation of further proteins that are involved in cell replication. The property itself attempts to determine: "the probability of their being i bound cyclin units at a particular instant in time, T".

Comparison of PRISM simulator results with PRISM

A PRISM experiment was ran where $N = 3$, i ranged from 0 to 3 and, t ranged from 0 to 40. The approximation parameter, ε was set to 0.01 and the confidence parameter, δ was set to 10^{-10}, meaning that 412,042 paths were generated. Each combination of i and t was enumerated into a set of properties that was evaluated concurrently.

The results are displayed in Figure 8.6. PRISM simulator results are indicated by vertical markers and PRISM results are indicated by horizontal markers.

Similar experiments were repeated for N ranging from 2 to 5, and the correlations were as strong for each case. From these results, it can be concluded that the PRISM simulator has been

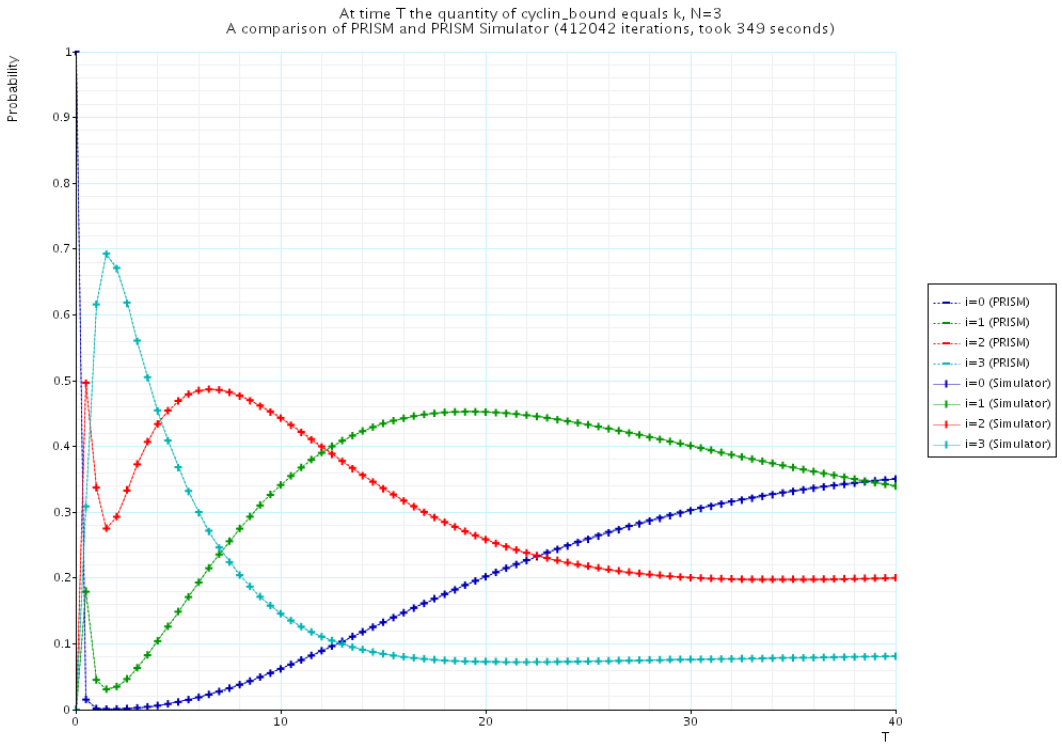


Figure 8.6: Comparison of PRISM and PRISM simulator results for the cell cycle example

successful in approximately verifying the behaviour of this model for input parameters that can be handled by PRISM.

Larger models

One of the key motivations for the PRISM simulator was that statistical sampling techniques do not need to enumerate the entire state space, and thus can handle larger models. In order to test this, the PRISM simulator was set up to consider the same property for models that were larger than could be handled by PRISM on a standard desktop PC.

For this experiment, $i = 0$ and T was varied from 20 to 60 in steps of 10. The chart displayed in Figure 8.7 shows the results for values of N varying from 1 to 16. For comparison purposes, the results calculated by PRISM are shown for N varying from 1 to 5.

From these results, the following observations can be made:

- When data is available for both PRISM and the PRISM simulator, the results are within the predefined error bounds.
- For this example, for larger values of N, the results calculated by the PRISM simulator appear to continue the trend calculated by PRISM.
- Without the PRISM simulator, it would not have been possible to observe some interesting patterns including where the different data series' cross each other and how the probability actually starts to increase with N for the $T = 20$ data series.

Taking into account that the correlation in results is strong for when PRISM can produce answers and that any trends appear to continue when calculated by the PRISM simulator, it could be concluded that the PRISM simulator is successful when applied to larger models. However, because there is still no source of comparison for the results of the larger models, such conclusions are dangerous.

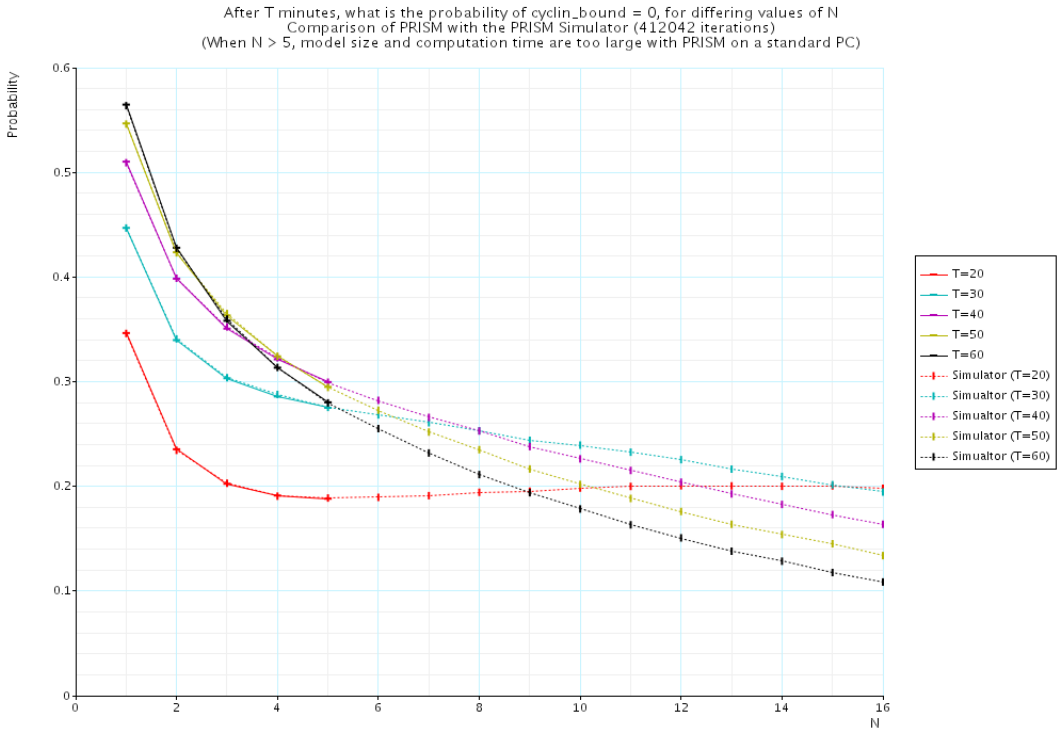


Figure 8.7: Continuation of trends calculated by PRISM by evaluating results with the PRISM simulator for larger models.

Chapter 9

Conclusions

This chapter summarises the achievements and deficiencies of the PRISM simulator project. A number of possible extensions to the project are also suggested in Section 9.3.

9.1 Summary of Project Acheivements

PRISM is a probabilistic model checker which allows the modelling and analysis of systems that exhibit probabilistic behaviour. This project added to the functionality of PRISM by providing a simulator, a tool which calculates and reasons about execution paths through probabilistic models.

9.1.1 The Simulator User Interface

The first part of the project was to create a simulator user interface, allowing users to manually or automatically generate and visualise paths through their models. For any given state of the model, the simulator calculates all possible transitions from that state, along with the probability or rate of that transition. The user is able to either select one of these transitions or specify that the simulator should make the selection. The states and transitions of a particular system model are calculated by loading in a model specified in the PRISM language, whose relevant features are supported in full for each model type.

Many features were incorporated to make the interface as useful as possible. For example, users are able to see which states satisfy certain logical predicates, whether execution paths satisfy path based formula and what the results of reward based formulae are over execution paths. To display a state satisfying a logical predicate, the user can either select a single state and query it's result, or they can select the predicate itself and request that the display of the execution path highlight all satisfying states.

The design for the user interface included many visual features in an attempt to make it more aesthetically pleasing but also to make it more usable. A sequence based view of the current execution path was designed to only display changes in variable values, an effect which was aimed at making it easier to spot patterns. Also, a loop detection feature was added, so that users could easily see where loops were starting and ending.

Feedback from users indicates that the tool is very useful as a debugger of PRISM models and properties, but also as a means of understanding the behaviour of models and even the behaviour of PRISM itself. Users even found it helpful as a teaching tool of probabilistic modelling and the PRISM language. For a wide variety of case studies the simulator user interface was evaluated as robust, responsive and scalable to large examples. Also, the visual features and interface layout received mainly positive feedback.

9.1.2 Approximate Model Checking

The second part of the project was to investigate and develop efficient algorithms to use statistical sampling techniques to approximate the probabilities or rewards of PRISM properties. The solution allows the user to specify the degree of approximation and the amount of confidence to which they require their results.

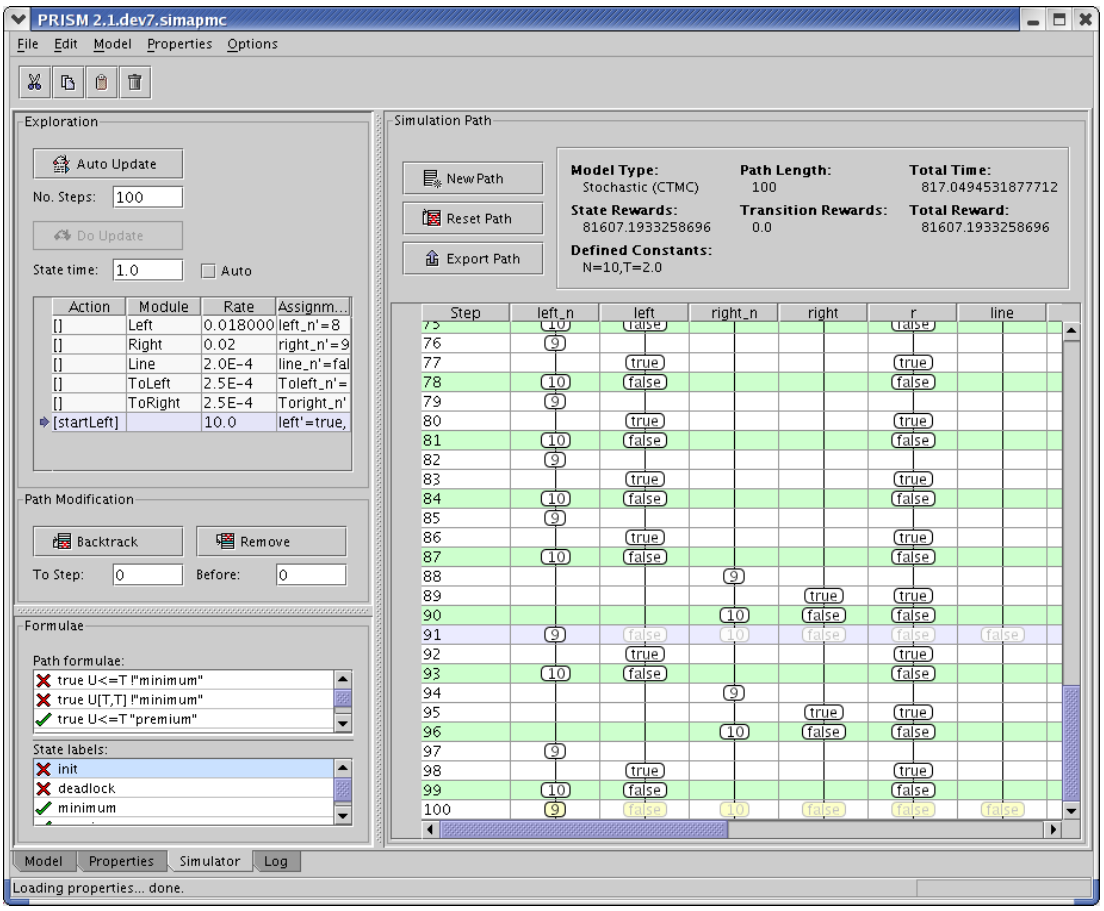


Figure 9.1: The simulator user interface integrated into PRISM

Several extensions to the simple *generic approximation algorithm* [6], provided a more efficient way of approximately verifying PRISM properties both independently and simulataneously. The solution ensured that paths were only as long as they needed to be, therefore saving computation time for models that had loops or paths that often satisfied a particular formula in just a few steps.

Despite the fact that these techniques can only be applied to a subset of the properties that can be handled by PRISM, this is due to reasons that could not be addressed by this project. These were discussed in Section 3.5.1.

These techniques were applied to property specifications of a wide range of discrete-time Markov chain and continuous-time Markov chain case studies. For all tests, results were produced that conformed to the approximation and confidence parameters. Also, results were produced for models whose state spaces are too large for PRISM to handle with a standard desktop PC. Although these results appear to continue the trends from the results calculated by PRISM on feasibily sized models, without a source of comparison for the larger models, it is difficult to conclude that the simulator produces reliable results for such large models. Nevertheless, the results are still quite convincing (see Figure 8.7).

9.1.3 Integration with PRISM

The simulator user interface and the statistical sampling techniques were integrated into the latest development version of PRISM as part of the user interface. The simulator user interface was included as an extra 'tab' and an extra button was provided to allow properties to be verified using the statistical sampling techniques, rather than PRISMs numerical engines. A screenshot of the simulator user interface is shown in Figure 9.1.

Furthermore, the simulator was integrated with PRISM's *experiments* user interface - a feature which allows properties to be verified against models using a range of input values. This

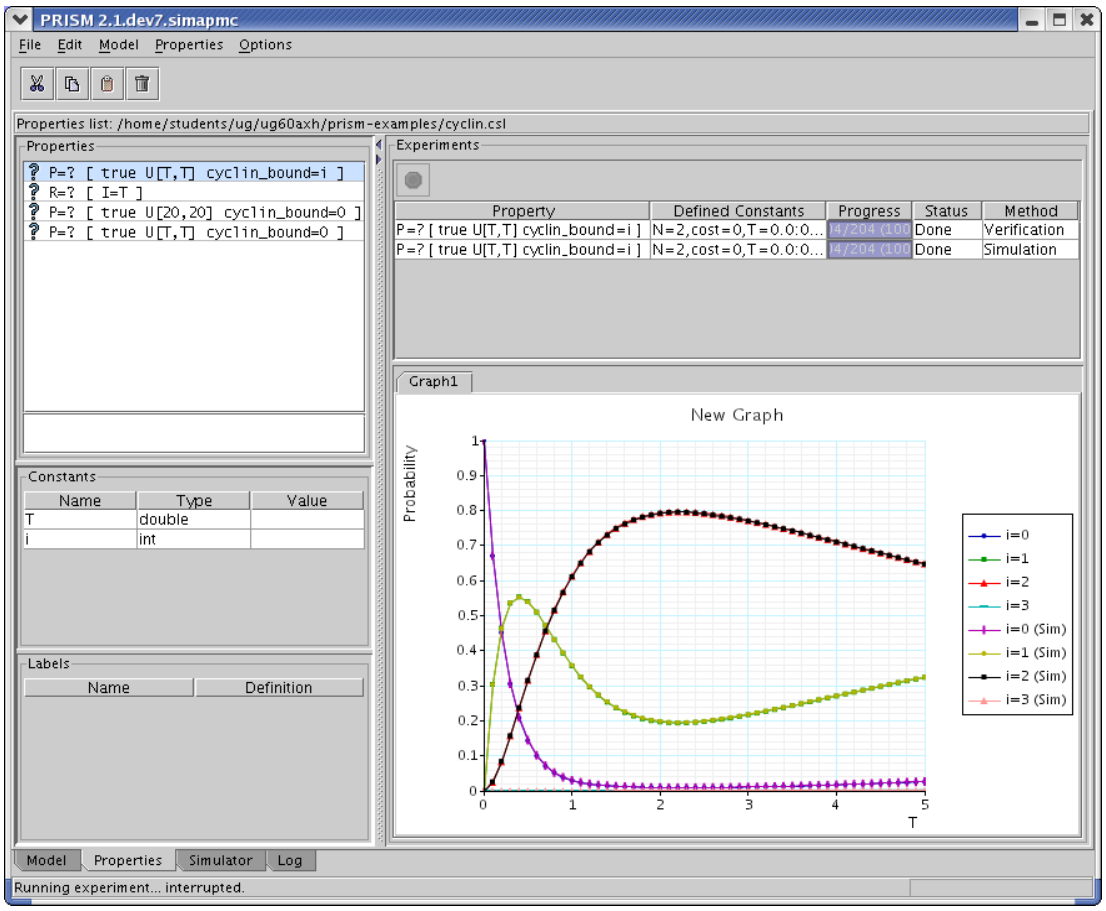


Figure 9.2: Integration of the PRISM simulator into the experiments user interface.

feature allows graphs of results to be generated quickly and compared with the results of PRISM's numerical engines. A screenshot of the integration of the simulator into the experiments user interface is shown in Figure 9.2.

9.2 Deficiencies

This section lists some of the main deficiences of the PRISM simulator together:

- The PRISM simulator does not support two features supported by the PRISM language. The first is that of specifying a process algebra to further customise the synchronisation of transitions. This feature was not included due to its relative complexity and the fact that it is rarely used in any of the available case studies. However, because this feature is not included, it is not possible to use the PRISM simulator for models that have been converted from the PEPA tool [1].
  The second feature that is not supported is that of specifying multiple initial states for the purposes of analysis. This feature was not included because it is relatively new and the benefits of including it as part of the simulator were unclear.

- The statistical sampling techniques are of little use for the analysis of the non-deterministic Markov decision process models. This is because their analysis requires analysis over all possible resolutions of non-determinism. With PRISM, it is possible to obtain the minimum and maximum probabilities over all possible resolutions, however, this cannot be done with a simulator without first enumerating the state space.

[1] This is a feature that is built into PRISM, where PEPA models are parsed into the PRISM abstract syntax tree and then built as PRISM models. The convertion of PEPA models tend to always use the process algebra feature.

- The statistical sampling techniques assumed that the storage of one sample path at a time would be sufficient to ensure that large PRISM models could be analysed. However, an example was found that had a very large number of variables, and required very long path lengths. It was not possible to store such a model in memory. The deficiency lies in the fact that it is not absolutely necessary to store an entire sample path at a time. In fact, it is only necessary to store the current state and a small set of states that could potentially form a loop at any one time.
- The statistical sampling techniques require unfeasibly large amounts of time to calculate very small probabilities. Unfortunately, many PRISM models require such small probabilities and so cannot be analysed by the PRISM simulator. This deficiency is really due to the complexity of the algorithm, and not the design of implementation of the project itself.
- For models with large numbers of variables, the menu, designed to allow users to hide / redisplay columns of the current execution path, is too big to fit onto the screen. This means that it is sometimes not possible to redisplay hidden variables.
- The feature that allows the user to highlight all states in the current execution path that satisfy a particular logical predicate is sometimes slow to the extent that it irritates users.

9.3 Extensions

This section suggests a number of possible extensions to the PRISM simulator. Many of these extensions were considered, but ultimately not implemented due to time constraints and complexity; some are to address specific deficiencies.

- Extend the coverage of the PRISM language to support the specification of process algebra to control the synchronisation of transitions. For more information, see the PRISM manual [2].
- Investigate the usefulness and feasiblity of extending the coverage of the PRISM langauge to support the specification of multiple initial states.
- Modify the data structures for the statistical sampling techniques so that they do not store entire sample paths. They should only store the current state and any previous states that are required for the purposes of loop detection.
- One possible approach to extend the range of properties that can be verified is to let the PRISM numerical engines calculate the set of states that satisfy particular nested formulae. Then, for each state in the execution path, σ, one would only have to check whether each state, s_i is in that set. This is the approach used in [9] and this extension is to integrate this into the PRISM simulator.
- A current development of the PRISM tool allows the user to specify models diagrammatically by use of a graphical model editor. Design and implement a link with the graphical model editor to allow graphical models to be visualised as part of the simulation user interface.
- Perform a more thorough analysis of applying statistical sampling techniques to Markov decision processes with the hope of producing better data structures, algorithms and results.

Appendix A

Project Proposal

A.1 Introduction

PRISM [2] is a probabilistic model checker, a tool which allows the modelling and analysis of systems that exhibit probabilistic behaviour. Modelling of a probabilistic system is done using the PRISM language, which can specify:

- **DTMCs** (Discrete-Time Markov Chains)
- **MDPs** (Markov Decision Processes)
- **CTMCs** (Continuous-Time Markov Chains)

The analysis of these models is done by specifying properties using the following temporal logics:

- **PCTL** (Probabilistic Computation Tree Logic) for DTMCs and MDPs
- **CSL** (Continuous Stochastic Logic) for CTMCs

PRISM builds a representation of the model into memory and performs numerical analysis to verify properties against it.

This project proposes to add to the functionality of PRISM by providing a simulator to allow further analysis of probabilistic models. Simulation involves the generation of *paths* through a model, which can either be generated manually or automatically. For each path, it is possible to determine whether certain properties have been satisfied, allowing the user to investigate PCTL and CSL specifications.

Simulation enables the user to manually explore the state space. This has the following benefits:

- It provides a good sanity check for users as they develop their models.
- Users may be able to spot errors and/or understand the behaviour of their model more easily if it is displayed visually.

In addition, simulation can be combined with statistical sampling techniques [9], to be used as an alternative to numerical analysis. This is particularly advantageous because a consequence of building certain models into memory is that build times and memory usage can be great, sometimes making numerical analysis impractical. Conversely, combined simulation and sampling techniques only require a single state of the model at a time, saving memory. Although these techniques yield less accurate results than numerical analysis, it is possible to obtain results within predefined error bounds.

This project will build upon an existing prototype simulator for PRISM DTMC models. This prototype allows automatic exploration of PRISM DTMC models, and some sampling based analysis of properties. Its main weakness is its limited functionality in terms of manual exploration, but it also suffers with speed, efficiency and usability problems.

A.2 Aims and Objectives

The aim of this project is to design and implement a simulator for PRISM. This shall be done by achieving the following objectives:

- Investigate, design and implement efficient data structures and algorithms for:
  - The automatic and manual generation of paths through each type of PRISM model;
  - The verification and analysis of PCTL and CSL properties over generated paths;
  - The computation of probabilities of certain PCTL and CSL properties, based on statistical sampling techniques.
- Design and implement an intuitive, but powerful graphical user interface for:
  - The automatic and manual exploration of PRISM models;
  - Allowing property verification using statistical sampling techniques, with the results also being made available for use in the PRISM experiments user interface.
- Investigate the use of error bounds and acceptance testing [6, 9] for use with the statistical sampling techniques and implement if feasible.

A.3 Extensions

The following extensions will also be considered:

- A current development of the PRISM tool allows the user to specify models diagrammatically by use of a graphical model editor. Design and implement a link with the graphical model editor to allow graphical models to be visualised as part of the automatic and manual exploration user interface.
- If the initial research into simulating MDP models does not yield good results, perform a more thorough analysis of the area with the hope of producing better data structures, algorithms and results.
- A current development of the PRISM tool allows paths to be given a cost or reward value. Integrate the idea of costs and rewards into the manual and automatic exploration algorithms and allow cost or reward based properties to be verified using the statistical sampling techniques.

A.4 Work Breakdown and Schedule

Project planning and initialisation All work concerning planning, proposal writing and project standards. Completion time: Late-September, 2004.

Requirements and Graphical User Interface Design All of the explicit requirements of the system should be collected. The user interface should be designed at the same time as the elicitation process to ensure that the specification is as complete and accurate as possible. Completion time: Mid-October, 2004.

1st iteration: Core data structures and algorithms The foundations of the project involve the design and implementation of core data structures and algorithms that will be used in future iterations. Completion time: Mid-November, 2004.

2nd iteration: Simulator user interface This phase involves research into the existing PRISM GUI, and then the production of an efficient class design and implementation of the new user interface. Completion time: Early-January, 2005.

3rd iteration: Properties user interface integration In this phase, the simulator's data structures and algorithms should be integrated into the existing 'properties user interface'. Completion time: Early-February, 2005.

1st Extension: Graphical module integration The simulator user interface should be extended to enable the users to see the current state of graphical modules diagrammatically.

2nd Extension: MDP simulation - further research and improvements Further research into MDP simulation should be performed with the aim of producing a more effective solution.

3rd Extension: Cost and Rewards Integration The idea of costs and rewards should be integrated into the simulator.

Dissertation Writing of the dissertation must have begun by 1st March, 2005 for completion on the 14th April, 2005. Any extra time before 1st March, 2005 can be used for the three project extensions. In addition, certain sections of the dissertation should be written as part of an ongoing documentation process.

A.5 Feasibility

All of the software and hardware needs for this project are met by the school.

Appendix B

UML Class Definitions

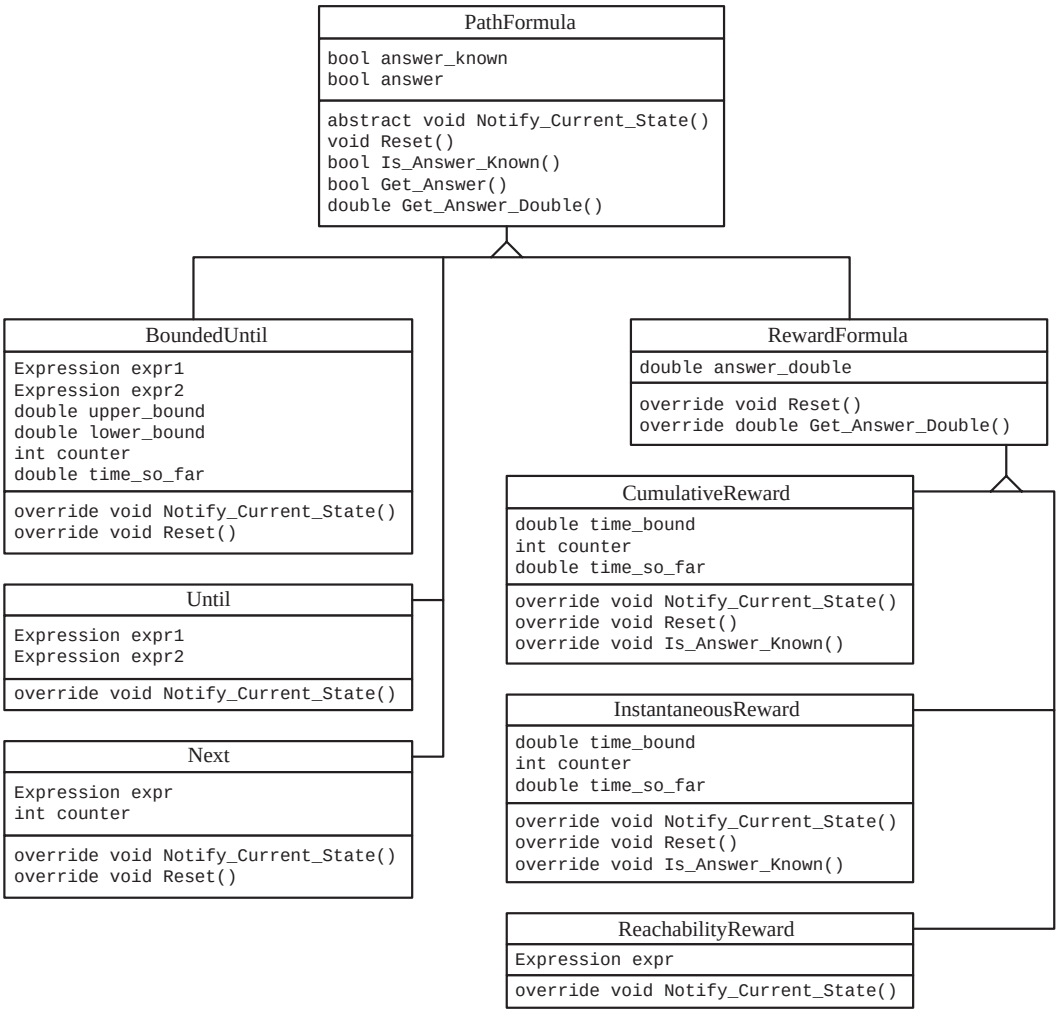


Figure B.1: UML diagram of the PathFormula inheritance hierarchy

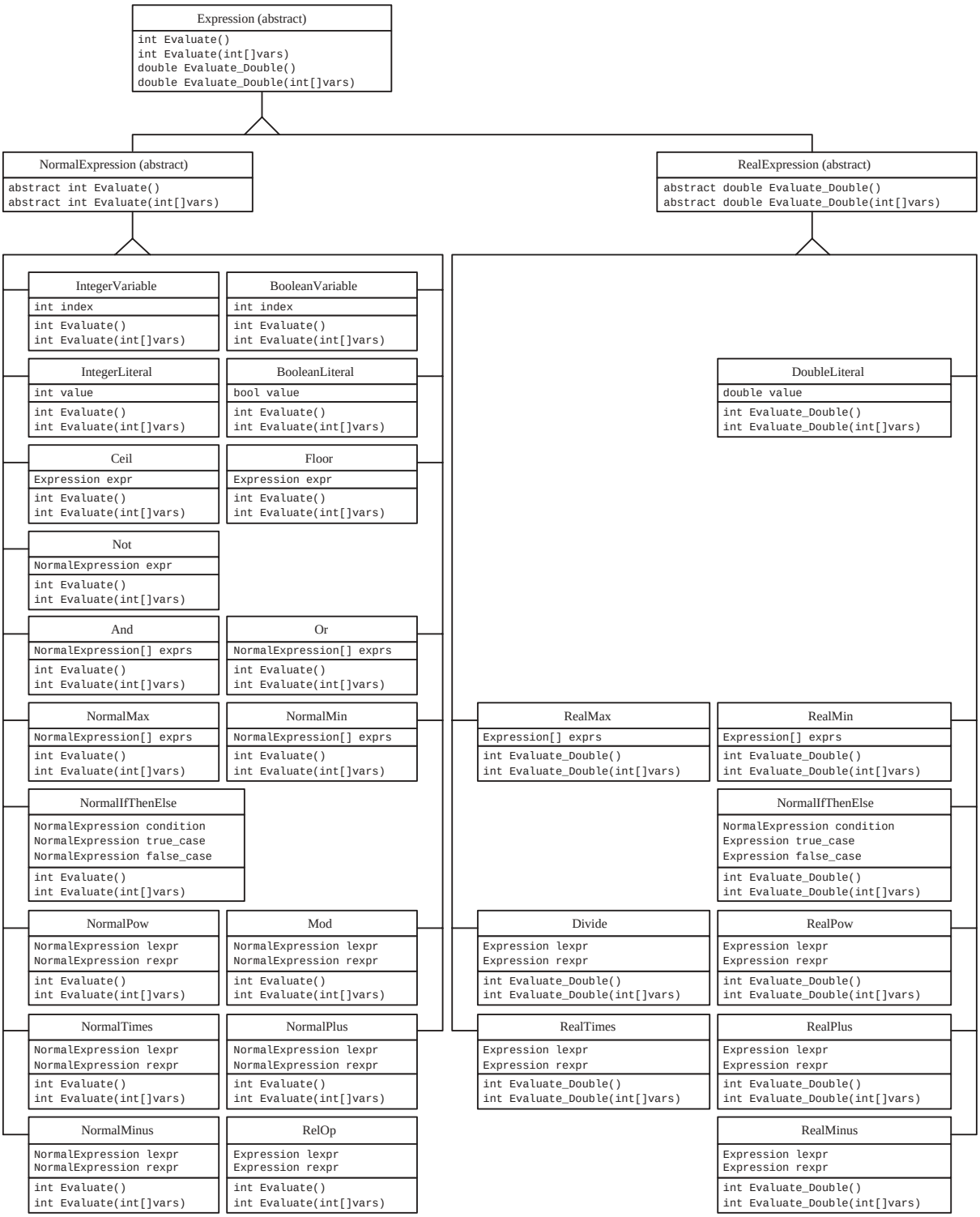


Figure B.2: UML diagram of the Expression inheritance hierarchy

Appendix C

Cell Cycle Case Study Model

```
// cell cycle control in eukaryotes
   // based on the stochastic pi caclulus model of Lecca and Priami

   // model is a ctmc
   stochastic

   // initial number of elements
   const int CYCLIN = 2 * N;
   const int CDK = N;
   const int CDH1 = N;
   const int CDC14 = 2 * N;
   const int CKI = N;

   const int N;

   // base rates of reactions
   const double R1  = 0.005;
   const double R2  = 0.001;
   const double R3  = 0.003;
   const double R4  = 0.5;
   const double R5  = 0.3;
   const double R6  = 0.005;
   const double R7  = 0.009;
   const double R8  = 0.009;
   const double R9  = 0.01;
   const double R10 = 0.017;
   const double R11 = 0.02;

   // module for including the base rates
   module base_rates

      [degp] true -> R1  : true;
      [degc] true -> R2  : true;
      [degd] true -> R3  : true;
      [lb] true -> R4  : true;
      [bb] true -> R5  : true;
      [cdh1r] true -> R6  : true;
      [pcdh1r] true -> R7  : true;
      [removep] true -> R8  : true;
      [removecki] true -> R9  : true;
      [donothing] true -> R10 : true;
      [bind] true -> R11 : true;
```

```
endmodule

// CYCLIN
module cyclin

    cyclin : [0..CYCLIN] init CYCLIN;
    cyclin_bound : [0..CYCLIN] init 0;
    degc : [0..CYCLIN] init 0;
    trim : [0..CYCLIN] init 0;
    dim : [0..CYCLIN] init 0;

    [lb] cyclin > 0 & cyclin_bound < CYCLIN
        → cyclin : (cyclin_bound' = cyclin_bound + 1) & (cyclin' = cyclin − 1);

    [degp] cyclin_bound > 0 & degc < CYCLIN
        → cyclin_bound : (degc' = degc + 1) & (cyclin_bound' = cyclin_bound − 1);
    [degd] cyclin_bound > 0
        → cyclin_bound : (cyclin_bound' = cyclin_bound);
    [bind] cyclin_bound > 0 & trim < CYCLIN
        → cyclin_bound : (trim' = trim + 1) & (cyclin_bound' = cyclin_bound − 1);

    [degc] degc > 0 → degc : (degc' = degc − 1);

    // rate dependent on the number of cyclin/cdk pairs which have a private channel
    [bb] trim > 0 & dim < CYCLIN → 1 : (dim' = dim + 1) & (trim' = trim − 1);

    [removecki] dim > 0 & cyclin_bound < CYCLIN
        → dim : (cyclin_bound' = cyclin_bound + 1) & (dim' = dim − 1);
    [donothing] dim > 0 → dim : (dim' = dim);

endmodule

// keeps track of the number of cyclin/cdk bounded pairs
// i.e. pairs what have a private channel
module counter

    bound1 : [0..min(CDK, CYCLIN)]; // pairs bound (cyclin has not bound with cki)
    bound2 : [0..min(CDK, CYCLIN)]; // pairs bound (cyclin bound with cki)

    [lb] bound1 < min(CDK, CYCLIN) → (bound1' = bound1 + 1);

    [degp] cyclin_bound > 0 & bound1 ≤ cyclin_bound →
        // probability cyclin which is bound takes part in the reaction
        bound1/cyclin_bound : (bound1' = bound1 − 1)
        // probability cyclin which is not bound takes part in the reaction
        + 1 − bound1/cyclin_bound : true;

    [bind] cyclin_bound > 0 & bound1 ≤ cyclin_bound & bound2 < min(CDK, CYCLIN) →
        // probability cyclin which is bound takes part in the reaction
        bound1/cyclin_bound : (bound1' = bound1 − 1) & (bound2' = bound2 + 1)
        // probability cyclin which is not bound takes part in the reaction
        + 1 − bound1/cyclin_bound : true;

    [degc] cdk_cat > 0 & bound1 + bound2 ≤ cdk_cat →
        // probability cdk which is bound (cyclin not bound to cki) takes part in the reaction
        bound1/cdk_cat : (bound1' = bound1 − 1)
        // probability cdk which is bound (cyclin bound to cki) takes part in the reaction
        + bound2/cdk_cat : (bound2' = bound2 − 1)
```

```
    // probability cdk which is not bound takes part in the reaction
    + 1 − (bound1 + bound2)/cdk_cat : true;

  [bb] bound2 > 0 → bound2 : (bound2′ = bound2 − 1);

endmodule

// CDK
module cdk

  cdk : [0..CDK] init CDK;
  cdk_cat : [0..CDK] init 0;

  [lb] cdk > 0 & cdk_cat < CDK → cdk : (cdk_cat′ = cdk_cat + 1) & (cdk′ = cdk − 1);

  [cdh1r] cdk_cat > 0 → cdk_cat : (cdk_cat′ = cdk_cat);
  [degc] cdk_cat > 0 & cdk < CDK → cdk_cat : (cdk′ = cdk + 1) & (cdk_cat′ = cdk_cat − 1);

  // rate dependent on the number of cyclin/cdk pairs which have a private channel
  [bb] cdk_cat > 0 → 1 : (cdk_cat′ = cdk_cat − 1);

  [removecki] cdk < CDK → 1 : (cdk′ = cdk + 1);

endmodule

// CDH1
module cdh1

  cdh1 : [0..CDH1] init CDH1;
  inact : [0..CDH1] init 0;

  [degp] cdh1 > 0 → cdh1 : (cdh1′ = cdh1);
  [cdh1r] cdh1 > 0 & inact < CDH1 → cdh1 : (cdh1′ = cdh1 − 1) & (inact′ = inact + 1);
  [removep] cdh1 > 0 → cdh1 : (cdh1′ = cdh1);

  [pcdh1r] inact > 0 & cdh1 < CDH1 → inact : (inact′ = inact − 1) & (cdh1′ = cdh1 + 1);

endmodule

// CDC14
module cdc14

  cdc14 : [0..CDC14] init CDC14;

  [pcdh1r] cdc14 > 0 → cdc14 : (cdc14′ = cdc14 − 1);

  [removep] cdc14 < CDC14 → (CDC14 − cdc14) : (cdc14′ = cdc14 + 1);

endmodule

// CKI
module cki

  cki : [0..CKI] init CKI;

  [degd] cki > 0 → cki : (cki′ = cki − 1);
  [bind] cki > 0 → cki : (cki′ = cki − 1);
```

```
endmodule


// rewards (number of proteins)
const int cost; // variable used to specify which reward we are interested in

rewards

    cost = 3 : cdh1;
    cost = 2 : cdc14;
    cost = 1 : cyclin_bound;

endrewards
```

Bibliography

[1] APMC web site. http://apmc.berbiqui.org/.

[2] PRISM manual. Available from www.cs.bham.ac.uk/~dxp/prism.

[3] PRISM web site. www.cs.bham.ac.uk/~dxp/prism.

[4] R. Downing. PrismSim: A tool for the simulation and verification of models specified using the PRISM system description language, 2002.

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

[6] R. Lassaigne and S. Peyronnet. Approximate verification of probabilistic systems. In H. Hermanns and R. Segala, editors, *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, volume 2399 of *LNCS*, pages 213–214. Springer, 2002.

[7] P. Lecca and C. Priami. Cell cycle control in eukaryotes: A BioSpi model. In *Proc. Workshop on Concurrent Models in Molecular Biology (BioConcur'03)*, Electronic Notes in Theoretical Computer Science, 2003.

[8] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.

[9] H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking: An empirical study. In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS. Springer, 2004. To appear.

[10] H. Younes and R. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In E. Brinksma and K. Larsen, editors, *Proc. 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 223–235. Springer, 2002.