SPECIFICATION AND VERIFICATION OF SYSTEMS USING MODEL CHECKING AND MARKOV REWARD MODELS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Farrel Lifson

May 2004

Supervised by

Prof. Pieter S. Kritzinger

Abstract

The importance of service level management has come to the fore in recent years as computing power becomes more and more of a commodity. In order to present a consistently high quality of service systems must be rigorously analysed, even before implementation, and monitored to ensure these goals can be achieved. The tools and algorithms found in performability analysis offer a potentially ideal method to formally specify and analyse performance and reliability models.

This thesis examines Markov reward models, a formalism based on continuous time Markov chains, and it's usage in the generation and analysis of service levels. The particular solution technique we employ in this thesis is model checking, using Continuous Reward Logic as a means to specify requirement and constraints on the model. We survey the current tools available allowing model checking to be performed on Markov reward models. Specifically we extended the Erlangen-Twente Markov Chain Checker to be able to solve Markov reward models by taking advantage of the Duality theorem of Continuous Stochastic Reward Logic, of which Continuous Reward Logic is a sub-logic.

We are also concerned with the specification techniques available for Markov reward models, which have in the past merely been extensions to the available specification techniques for continuous time Markov chains. We implement a production rule system using Ruby, a high level language, and show the advantages gained by using it's native interpreter and language features in order to cut down on implementation time and code size.

The limitations inherent in Markov reward models are discussed and we focus on the issue of zero reward states. Previous algorithms used to remove zero reward states, while preserving the numerical properties of the model, could potentially alter it's logical properties. We propose algorithms based on analysing the continuous reward logic requirement beforehand to determine whether a zero reward state can be removed safely as well as an approach based on substitution of zero reward states. We also investigate limitations on multiple reward structures and the ability to solve for both time and reward.

Finally we perform a case study on a Beowulf parallel computing cluster using Markov reward models and the ETMCC tool, demonstrating their usefulness in the implementation of performability analysis and the determination of the service levels that can be offered by the cluster to it's users.

Acknowledgements

This dissertation was completed with the assistance of a number of people. My generous thanks must be given to

- Prof P.S. Kritzinger, who has provided fine supervision and gracious financial assistance.
- Prof Joost-Pieter Katoen and Prof Holger Hermanns, of the University of Twente and Saarland University respectively, for their appreciated generosity in allowing me to visit the University of Twente for three months and who provided invaluable assistance and answers to questions.
- Prof Markus Siegle of the University of Erlangen-Nuremburg, who provided excellent assistance and generous use of his time during his short visit to UCT.
- Mark Horner and Bruce Becker, of the Department of Physics at UCT, for their much appreciated assistance.
- My parents for their continued assistance and support.
- Mia, for her almost limitless understanding and much needed and appreciated support.
- All the members of the Data Network Architecture Lab and the M.Sc lab during my time in Room 300, for providing a friendly working atmosphere and an ideal sounding board off which to bounce ideas.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With computing becoming more and more of a commodity, users have come to expect a constant level of high service and availability. This has necessitated service providers to carefully manage and plan their implementation in order to ensure that the maximal service is provided to the consumer, while at the same time lowering the cost to the provider. The levels of service that can be offered often need to be determined even before implementation in order to be able provide performance and reliability estimates to users upfront so that these can be accounted for in the design and implementation of applications the user might require.

The methods used in the past to determine service levels are extremely varied ranging from tested methods using models such as queueing networks to outright estimation. Some of the known problems[Lib01] encountered when determining service levels include

- Lack of clear understanding of the objectives
- No visible benefit
- Unrealistic expectations

All of these problems can be overcome through the usage of a clearly defined specification language that is easily verified. We believe the close intertwining between Continuous Reward Logic (CRL) and Markov Reward Models (MRM) provide an ideal solution to easily model a system, using a variety of techniques, and to verify that the requirements are met in a convenient and powerful manner.

1.1.1 Service Level Management

Service level management is defined as the set of of people and systems that allow the organisation to ensure that service level agreements are being met[Con03]. Performability analysis methods and techniques can play a large part in ensuring that these goals are met by allowing the system designer to quickly and accurately gauge the performance and reliability properties before implementation.

At the core of service level management is the service level agreement (SLA), a contract between two parties detailing the level of service one party can expect from the other. These SLAs are expressed usually as a function of performance or reliability, specifying definite goals. However the models these goals are applied to are often probabilistic in nature and may not be able to meet these deterministic requirements in every instance, although they may meet the requirements in the majority of cases.

Formal methods based on continuous time Markov chains have proven successful in assisting in determining service level agreements[KWA99, MA98, FF97]. The emergence of performability logics coupled with the ability to perform model checking on models using these logics has added another dimension to performability analysis, allowing systems to be specified and verified in a strict, formally defined manner.

Figure 1: The Telecommunications Management Network Model

In the telecommunications management network model[Con03], shown in Figure 1, service management is placed logically above the network and element management layers. While there is abundant research investigating the application of service level agreements on these lower levels [CFK+02, HW02], our research and investigation will be more concerned with the specification and verification of the systems properties not on those of it's individual components. This however does not mean that we can ignore the behaviour and properties of the components which reside in the network and element layers as the final behaviour of the system is ultimately an aggregate of the behaviour of these components.

Figure 2 shows the service level management elements in a telecommunications network as defined by Bouilley, Mira and Ramakrishnan[BMR02]. The area where we will be focusing on is the *off-line* area on the left of the dashed line, which is analogous to the design phase of the system. This subsection of the service management process is the most relevant to us as it is during the design and SLA planning phase where the initial system model will be determined. Once we have passed into the real-time SLA management phase the need for models and requirements is diminished significantly.

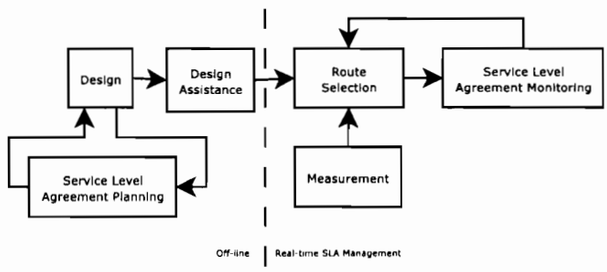

Figure 2: Service Level Management Elements in a Telecommunications Network

By taking advantage of the various abstraction facilities in both CRL and MRM we will be able to show the usefulness of MRM in determining system wide service levels while still taking into account the behaviour of components on the network and element levels. To facilitate this we have extended the Erlangen-Twente Markov Chain Checker (ETMCC)[HKMKS99] tool to handle Markov reward models and have also implemented a production rule MRM generator.

1.2 Previous Work

1.2.1 Limitations of Model Checking MRM

Kateon et al.[BHHK00b] note that for the Duality Theorem(Section 2.4.2) to be applicable the MRM can only have positive non-zero rewards. This requires any zero reward states in the MRM to be removed before any transformation can take place.

Beaudry[Bea78] originally proposed an algorithm that would render every zero reward state absorbing. This is not always desirable as it can affect the performance properties of the model. Ciardo et al.[CMST90] devised an algorithm which would wholly remove zero reward states from the model which, while preserving the performance properties of the model, can affect the logical properties. Our proposed refinement combines both methods by analysing the CRL requirement beforehand to decide whether the affected zero reward state should be removed or made absorbing.

1.2.2 Performability Analysis Using CRL and MRM

There have been a number of past case studies performed using only MRM[ST88, QS96], however the relatively recent appearance of model checking algorithms and tools has meant that there has been little practical use of CRL in performability analysis.

Despite being especially suited to the determination and specification of service levels the examples and case studies performed using CRL and CSL that we are aware of[HKMKS00, HKMKS99, HCH+02, YKNP04] focus more on showing the correctness and performance of the algorithms and less so on the actual usage of CRL and MRM in a practical setting. The relative scarcity of model checking tools available (which we cover in more detail in Chapter 3) when compared to the number of tools available for the solution of continuous time Markov chains (CTMC) and the lack of integration of these tools in larger performability analysis frameworks has also contributed to this.

1.3 Dissertation Contribution

In this dissertation we investigate the usage of MRM and CRL in the determination and verification of performability service levels. We observe a number of present restrictions on the model, logic and available algorithms used in the verification of service requirements.

For the zero reward reward limitation we propose analysis of the CRL requirement beforehand to determine which states can be safely removed without affecting the logical properties of the model. We also propose a method using substitution of zero reward states, a method which introduces error to the model which we quantify via simulation.

We also show that due to the independence of nested CRL statements that it is possible to have multiple reward structures present in a MRM with no additional solution techniques needed.

We demonstrate the usefulness of MRM and CRL in specifying service levels by performing a case study on a production Beowulf class parallel cluster. We successfully were able to use MRM and CRL to demonstrate the significant affect power outages have on the performability of the cluster as well as in determining the budgetary requirements of the cluster.

1.4 Dissertation Outline

The rest of this dissertation is laid out as follows:

Chapter 2 introduces Markov reward models, Continuous Stochastic Logic, Continuous Reward Logic as well as the techniques and algorithms developed to be able to perform model checking using these formalisms.

Chapter 3 investigates the currently available model checking tools, as well as the differing methods they employ in model checking performability requirements.

Chapter 4 covers various available modelling formalisms and we investigate the ability of these formalisms to model Markov reward models. We also describe a simple production rule system to generate Markov reward models using a high level language interpreter.

Chapter 5 tackles the limitations inherent in the algorithms used to model check Markov reward models. We present two proposals to combat the difficulty of model checking with zero reward states. The ability, or rather lack thereof, to solve for both time and reward is also discussed.

Chapter 6 covers a case study carried out on a Beowulf class cluster currently used by the Department of Physics at UCT. We investigate the usage of CRL and MRM in determining service levels, in terms of both performance and financial cost, we can expect from the cluster during it's current and planned future operation.

Chapter 7 concludes the dissertation and we examine possible future areas of interest.

Chapter 2

Markov Reward Models and Performability Logics

2.1 Introduction

In this chapter we introduce the two fundamental concepts on which we rely in order to perform performability analysis. Markov reward models are the modelling formalism we employ to model the performability properties of the system in question. To this model we then apply requirements specified using the Continuous Reward Logic performability logic. Both the modelling and requirement components are based on known mathematical algorithms and analysis techniques and in this chapter we will detail these algorithms and their usage in the solution of Markov reward models.

2.2 Continuous Time Markov Chains

Markov Processes are a class of stochastic process, developed by Markov in 1907, whose conditional probability function is such that the probability of the process having a certain value is dependent only on what the value of the process was immediately before. Mathematically this is expressed as

$$P[\mathcal{X}(t) = x | \mathcal{X}(t_n) = x_n, \mathcal{X}(t_{n-1}) = x_{n-1}, \ldots, \mathcal{X}(t_0) = x_0] = P[\mathcal{X}(t) = x | \mathcal{X}(t_n) = x_n]$$

The expression is known as the *Markov property*, and any stochastic process which obeys it is a Markov process. The property can be informally explained as saying that the process has no memory and that the future value of the process depends solely on the current value.

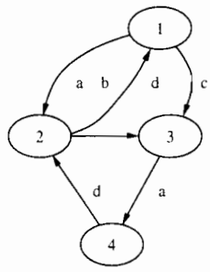

Figure 3: A simple Continuous Time Markov Chain

With this definition we introduce the concept of a Continuous Time Markov Chain (CTMC). A CTMC is a Markov process described using *states* and *transitions* between the states and with the probability density function defined on the continuous real line (as opposed to Discrete Time Markov Chains which are defined at discrete time steps).

We will define a *labelled* CTMC as a tuple $(S,\mathbf{R},L)$ with

- S: a finite set of *states* that the CTMC can be in
- $\mathbf{R}$: $S \cdot S \rightarrow \mathbb{R}$, the *rate matrix*, which assigns to each transition a rate
- L: $S \rightarrow 2^{AP}$, the *labelling function* which maps to each state a label, comprised of atomic propositions with AP a finite fixed set of these atomic propositions.

In Figure 3 we present a simple CTMC whose tuple is composed of the following

- $S = \{1, 2, 3\}$
- $\mathbf{R} = \left\{ \begin{array}{cccc} 0 & a & c & 0 \\ d & 0 & b & 0 \\ 0 & 0 & 0 & a \\ 0 & d & 0 & 0 \end{array} \right\}$
- $L(s) = s$ with $s \in S$

We must note that in our definitions of CTMC we allow $\mathbf{R}(s, s) > 0$. This allows the system to occupy the same state before and after a transition is taken. This does not affect the transient or steady-state behaviour of the system but allows the usual interpretation of linear-time temporal operators [BHHK03].

If a transition exists between s and s' then $\mathbf{R}(s,s') > 0$ and the probability of that this transition is triggered within t units of time is $1 - e^{-\mathbf{R}(s,s')t}$. A *race condition* occurs when there is more than one state s' such that $\mathbf{R}(s,s') > 0$. The probability of a particular transition being taken within t units of time and being the transition taken before any other is taken is

$$\mathbf{P}(s,s',t) = \frac{\mathbf{R}(s,s')}{E(s)}(1 - e^{-E(s)t})$$

Where $E(s)$ is the *total exit rate* from state s, specified as $E(s) = \sum_{s' \in S} \mathbf{R}(s,s')$. We also define the *transition matrix* $\mathbf{P}(s,s')$ as $\mathbf{R}(s,s')/E(s)$ as well as an *initial distribution*, a function M_0, which assigns to each state s an initial probability of being in that state. Formally $M_0 : S \to [0,1]$ such that $\sum_{s \in S} M_0(s) = 1$. For certain cases of CTMC the initial distribution has no affect on the steady state of the system[BK95] and this steady state distribution is independent of all initial probability distributions.

2.3 Markov Reward Models

A Markov Reward Model (MRM) is a CTMC with an associated reward structure. The reward structure, ρ, is a function that assigns to each state s some numerical $\rho(s)$. When residing for t time units in state s, we acquire a total reward of $\rho(s) \cdot t$.

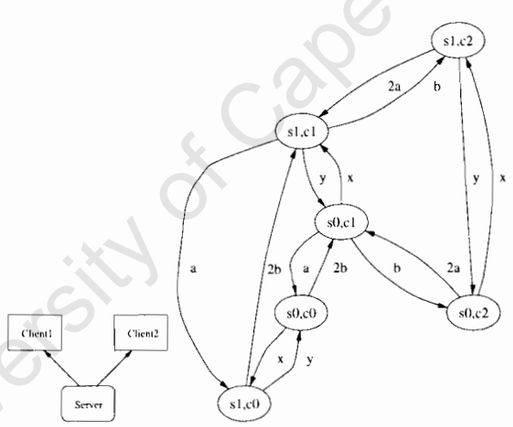

Figure 4: Simple client-server system and CTMC extracted from it

In Fig. 4 we show a simple client-server system with two clients and a server. The server sends to each client a job to process. The clients fail at rate a, and are repaired at rate b while the server fails at rate y and is repaired at rate x. In Figure 4 we also show a CTMC derived for this model.

In each state we specify what properties the state satisfies (ie $s1, c1$ - that 1 server and 1 client are available). To make the model a MRM we add the following reward structure:

$$\rho(s) = \begin{cases} 2 & \text{if } s \models c2 \wedge s1 \\ 1 & \text{if } s \models c1 \wedge s1 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The reward structure is a simple one meant to model the availability of the system. If we assume that each client processes one job per time unit then a reward of 2 means both clients are up and processing jobs from the server. From the reward structure it is also apparent that if the server is down then no jobs are processed even if both clients are available.

2.4 Performability Logics

2.4.1 Continuous Stochastic Logic

Associated with MRM are logics which are used to specify desired properties of the system. The logics we will look at are Continuous Reward Logic (CRL) and Continuous Stochastic Logic (CSL). CRL is a logic that deals purely with rewards while CSL is a logic that deals with time. The two logics are in fact sub-logics of Continuous Stochastic Reward Logic (CSRL), a logic which combines both reward and time.

Syntax

CSL belongs to the class of branching time temporal logics and is based on the pre-existing (equally named) logic CSL[ASSB96] described by Aziz et al. as well as using the probabilistic $\mathcal{P}$-operator found in PCTL[HJ94]. In CSL we extend these logics by applying bounds to both the until ($\bigcup$) and next (X) operators. The following define what is allowed as a path- and state-formula:

- tt (the atomic proposition that is true in all states) is a state-formula
- each atomic proposition $a \in AP$ is an a state-formula
- if Φ and Ψ are state-formulas then so are their negation ($\neg\Phi$) and intersection ($\Phi \wedge \Psi$)
- if Φ is a state-formula, then so is the steady state operator applied to it - $\mathcal{S}_{\trianglelefteq p}(\Phi)$ where $\trianglelefteq \in \{\leq, \geq\}$ and $p \in [0, 1]$
- if φ is a path-formula then the the probabilistic operator applied to it is a state-formula - $\mathcal{P}_{\trianglelefteq p}(\varphi)$

- if Φ and Ψ are state-formulae, then $\mathsf{X}^I\Phi$ and $\Phi \bigcup^I \Psi$ are path formulae, with $I \subseteq \mathbb{R}$

In usage $\mathcal{S}_{\trianglelefteq p}(\Phi)$ states that the steady-state probability for a Φ-state satisfies the condition $\trianglelefteq p$. Similarly, $\mathcal{P}_{\trianglelefteq p}(\varphi)$ states that the probability measure of paths which satisfy φ, also meet condition $\trianglelefteq p$. The $\mathcal{P}$ operator can also be used to replace the CTL path quantifiers $\exists$ (there exists) and $\forall$ (for all). In most cases $\exists\varphi$ (there exists a path for which φ holds) can be written as $\mathcal{P}_{>0}(\varphi)$ while $\forall\varphi$ (for all paths φ holds) can be expressed as $\mathcal{P}_{\geq 1}(\varphi)$. Due to fairness considerations these rules are not generally applicable[BK98].

Further CSL notation shortcuts include $\Diamond\Phi$ and $\Box\Phi$. $\Diamond\Phi$ represents the path-formula $tt \bigcup \Phi$ (i.e. eventually Φ holds) while $\Box\Phi$ is shorthand for $\neg\Diamond\neg\Phi$ (i.e. invariantly Φ holds).

Semantics

Let $\mathcal{M} = \{S, \mathbf{R}, L\}$ be a CTMC with labels in AP. We define the meaning of a CSL statement by means of $\models$, the satisfaction relation between $\mathcal{M}$, one of its states s and a state-formula Φ. $s \models \Phi$ iff Φ is valid in s. We define $Sat(\Phi) = \{s \in S | s \models \Phi\}$ the the rest of the satisfaction relation as follows

$$
\begin{array}{llll}
s \models \mathrm{tt} & \forall s \in S & s \models \Phi \wedge \Psi & \iff s \models \Phi \wedge s \models \Psi \\
s \models a & \iff a \in L(s) & s \models \mathcal{S}_{\trianglelefteq p}(\Phi) & \iff \pi(s, Sat(\Phi)) \trianglelefteq p \\
s \models \neg\Phi & \iff s \not\models \Phi & s \models \mathcal{P}_{\trianglelefteq p}(\varphi) & \iff Prob^{\mathcal{M}}(s, \varphi) \trianglelefteq p
\end{array}
$$

where $Prob^{\mathcal{M}}(s, \varphi) = \Pr_s\{\sigma \in Path^{\mathcal{M}} | \sigma \models \varphi\}$ which is the probability measure of all paths starting in s which satisfy φ. We can also define the meanings of the path-formulae in a similar fashion

$$
\begin{array}{ll}
\sigma \models \mathsf{X}^I\Phi & \iff \sigma[1] \text{ is defined and } \sigma[1] \models \Phi \wedge \lambda(\sigma, 0) \in I \\
\sigma \models \Phi \bigcup^I \Psi & \iff \exists t \in I \text{ s.t. } \sigma@t \models \Phi \wedge \forall t' \in [0, t) \text{ s.t. } \sigma@t' \models \Phi
\end{array}
$$

The use of CRL and CSL as separate logics, is more common up until now as verification of CSRL is numerically complex[HCH+02]. The operators of CSRL as well as some examples of CSRL,CSL and CRL are shown in Figure 5.

2.4.2 Duality

CSRL has a property known as the *Duality theorem*[BHHK00b] which allows us to transform the MRM in such a way that we can swap the reward and time constraints of the logic. The informal reasoning behind this is that we can regard the progress of time as the accumulation of rewards and the same in reverse.

State-formulas $\Phi ::= a | \neg\Phi | \Phi \vee \Phi | \mathcal{S}_{\trianglelefteq p}(\Phi) | \mathcal{P}_{\trianglelefteq p}(\varphi)$

$\mathcal{S}_{\trianglelefteq p}(\Phi)$**:** probability that Φ holds in steady state is $\trianglelefteq p$

$\mathcal{P}_{\trianglelefteq p}(\varphi)$**:** probability that paths fulfill φ is $\trianglelefteq p$

Path-formulas $\varphi ::= \mathsf{X}^I_J \Phi | \Phi \bigcup^I_J \Phi$

$\mathsf{X}^I_J \Phi$**:** next state reached at time $t \in I$ and reward with $r \in J$ and fulfils Φ

$\Phi \bigcup^I_J \Psi$**:** Φ holds along the path until Ψ holds at time $t \in I$ and reward $r \in J$

CSRL: $\mathcal{P}_{>0.95}(\mathsf{X}^{[0..20]}_{[10..20]} c1 \vee c2)$ - The probability that at least one client is running within 20 time units and with a total reward of between 10 and 20 points is > 0.95.
CSL: $\mathcal{P}_{<0.01}((c1 \vee c2) \bigcup^{[0..100)} (c0 \vee s0))$ - The chance of failure (no jobs being processed) before 100 time units is less than 0.01 percent.
CRL: $\mathcal{P}_{<0.03}(\mathsf{X}_{[0..50]} c1 \wedge s1)$ - The probability of one server failing while we have accumulated 50 rewards or less is less than 0.03 percent.

Figure 5: The syntax and examples of CSRL

From this we see that the verification of properties described in CRL do not need any other specialised techniques other than the ones needed to verify CSL, which we will detail in Section 2.5.

This Duality theorem allows us to verify a CRL statement by transforming the statement into an equivalent CSL statement and then verifying this statement against the modified CTMC. Note, however, that the transformation is restricted to non-zero rewards (if $\rho(s) = 0$ then $\mathbf{R}(s, s')/\rho(s)$ will be undefined). We investigate this later in the dissertation in Section 5.2.2.

To transform our MRM into a CTMC that can be checked using CSL verification methods we do the following:

1. Transform MRM $\mathcal{M} = ((S, \mathbf{R}, L), \rho)$ into $\mathcal{M}^{-1} = ((S, \mathbf{R}', L), \rho')$ with
   - $\mathbf{R}'(s, s') = \mathbf{R}(s, s')/\rho(s)$ - rescale transition rates by reward
   - $\rho'(s) = 1/\rho(s)$ - invert the reward structure
2. Transform state formula Φ into Φ^{-1} by swapping reward and time bounds : $\mathsf{X}^I_J \rightarrow \mathsf{X}^J_I$ and $\bigcup^I_J \rightarrow \bigcup^J_I$

The duality theorem then states:

$$Sat^{\mathcal{M}}(\Phi) = Sat^{\mathcal{M}^{-1}}(\Phi^{-1}) \tag{2}$$

2.5 Model Checking CSL

To begin model checking we need two components Φ, the constraint we wish to test specified in CSL/CRL, and M the underlying CTMC that describes the system. A variety of algorithms are utilised in order to best ensure that a solution is found as efficiently as possible. The numerical analysis of CTMCs have been studied extensively and numerous algorithms[BKH99, BHHK00a] have been devised in order to solve CSL requirements. This section summarises the techniques devised in *Model-Checking Algorithms for Continuous Time Markov Chains* by Baier, Haverkort et al.[BHHK03].

The algorithms required depend on on the CSL operator being checked. For the $\mathsf{X}-operator$ simple matrix-vector multiplication is used, for the unbounded $\bigcup$ and $\mathcal{S}$-operators the solution of linear equations is needed and for $\bigcup_{\geq p}$ the solution of systems of Volterra integral equations needs to be found.

2.5.1 Non-Stochastic Operators

For non-probabilistic operators the algorithms needed are the same as that for Continuous Temporal Logic[CES86], as CSL is merely CTL with added probalistic and stochastic operators.

To check whether $s \models \mathcal{S}_{\trianglelefteq p}(\Phi)$ we first need to determine the set of states, $Sat(\Phi)$, that will satisfy our state formula Φ. This is using the following algorithm

- $Sat(a)$ is the set of states labelled with atomic proposition a
- $Sat(\Phi \vee \Psi)$ is $Sat(\Phi) \bigcup Sat(\Psi)$
- $Sat(\neg\Phi)$ is $S - Sat(\Phi)$
- $Sat(\exists \mathsf{X}\Phi)$ is the set of states that can move directly to $s \in Sat(\Phi)$
- $Sat(\Phi \bigcup \Psi)$is computed as follows
  - $S^0 = Sat(\Psi)$
  - $S^1 = S^0 \bigcup \Phi$-states that can move directly to S^0
  - $S^2 = S^1 \bigcup \Phi$-states that can move directly to S^1
  - $...until S^{k+1} = S^k$

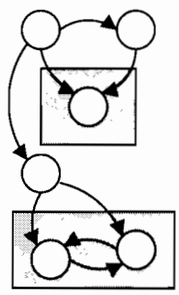

Figure 6: A CTMC with BSCCs indicated in the grey shaded ares

2.5.2 Steady State Measures

When computing the steady state measure of a CTMC, it is quite often apparent that certain states will have a residence probability that tends to 0 as $t \to \infty$, and so they become uninteresting when dealing with steady state measures. A *bottom strongly connected component* (BSCC) of a directed graph is a subgraph, that once entered, can not be exited from. By computing the BSCC of a a CTMC we can potentially remove a significant proportion of the states from consideration. Figure 6 shows two such BSCCs in a CTMC. Note that a BSCC can be composed of only one state.

We can reduce any CTMC to a directed graph by treating states as vertices and drawing an edge from s to s' iff $\mathbf{R}(s, s') > 0$. We can then describe the BSCC of a CTMC as a set of states, B, such that $Reach(s) \subseteq B \forall s \in B$ where $Reach(s)$ is the set of states reachable from s. A state s will satisfy $\mathcal{S}_{\trianglelefteq p}(\Phi)$ iff $\pi(s, Sat(\Phi)) \trianglelefteq p$.

Each BSCC, if separated from the original CTMC, describes a CTMC which is also strongly connected. We can then use the fact that the steady state distribution for such a CTMC $\underline{\pi}^B$ is independent of it's initial distribution. The following equation describes how to compute this steady state distribution.

Let $\mathcal{M}$ be a CTMC $(S, \mathbf{R}, L)$ with $s \in S, S' \subseteq S$ and a set of BSCCs $B(\mathcal{M})$. As $t \to \infty$ each path σ will at some point enter a BSCC with probability 1. The probability of being in a BSCC is therefore the probability of entering it which we can express as

$$\{Pr_s\sigma \in Path | \exists t \text{ s.t. } \sigma(t) \in B\} = \{Pr_s\sigma \in Path | \sigma \models \Diamond at_B = Prob(s, \Diamond at_B)\} \tag{3}$$

The steady state probability of some state in $B \bigcap S'$ is equal to the probability of eventually ending up in B multiplied by the steady state probability of being in a state in $B \bigcap S'$ if starting somewhere in B. Using the law of total probability we can sum over all possible BSCCs and obtain the steady state probability of being in some state in S'. Formally, this can be expressed as

$$\pi(s, S') = \sum_{B \in B(\mathcal{M})} \left(Prob(s, \Diamond at_B) \cdot \sum_{s' \in B \bigcap S'} \pi^B(s') \right) \tag{4}$$

To check whether $s \models \mathcal{S}_{\trianglelefteq p}(\Phi)$ we first compute recursively the set of states that satisfy Φ as well as the set of BSCCs of the CTMC under consideration. For each BSCC, B, such that $s' \in B$ and $s' \models \Phi$, we compute the steady state probabilities to determine $\pi^B(s')$. These steady states are computed using standard techniques to solve linear equations that characterise the steady state properties. This linear equation is

$$\sum_{s \in B, s \neq s'} \pi^B(s) \cdot \mathbf{R}(s, s') = \pi^B(s') \cdot \sum_{s \in B, s \neq s'} \mathbf{R}(s, s') \text{with} \sum_{s \in B} \pi^B(s) = 1 \tag{5}$$

where π^B is a vector of size $|B|$, except when $B = s$ which implies that $\pi^B(s) = 1$. Also note that states not in any of the BSCCs have steady state probability 0, regardless of the initial distribution. The probability of eventually reaching a BSCC, B, from some state s is given as

$$Prob(s, \Diamond at_B) = \begin{cases} 1 & \text{if } s \models at_B \\ \sum_{s'} \mathbf{P}(s, s') \cdot Prob(s', \Diamond at_B) & \text{otherwise} \end{cases} \tag{6}$$

When the CTMC is strongly connected, a single BSCC occurs (consisting of all the states in the CTMC) and the entire procedure reduces down to solving the steady state probabilities for all Φ-states in the CTMC.

2.5.3 Probabilistic Path-Measures

Unlike state measures, the path measures are characterised by properties which start in a certain state (or set of states) and only traverse paths which satisfy certain properties. As the basis for these measures we use $Prob(s, \varphi)$, the probability measure for the set of paths that start in s and fulfill φ.

Proposition 1 For the time bounded next operator (X^I) the following measure is defined. $s \in S$, interval $I \subseteq \mathbb{R}_{\geq 0}$, Φ a CSL state formula and $E(s)$ is the total exit rate from state s.

$$Prob(s, \mathsf{X}^I \Phi) = \left(e^{-E(s) \cdot \inf I} - e^{-E(s) \cdot \sup I} \right) \cdot \sum_{s' \models \Phi} \mathbf{P}(s, s') \tag{7}$$

The proof of this statement relies on the definition of Borel spaces over paths[BHHK03] and that the probability to take an outgoing transition from state s at some time $t \in I$ is

$$\int_I E(s) \cdot e^{-E(s)\cdot t} dt = e^{-E(s)\cdot \inf I} - e^{-E(s)\cdot \sup I} \tag{8}$$

By multiplying the above result with the probability of reaching a ϕ-state in one step described by $\sum_{s' \models \Phi} \mathbf{P}(s, s')$, we end up with the required formula.

Using this result we can obtain the following algorithm to compute $Sat(\mathcal{P}(\mathsf{X}\Phi))$. The set $Sat(\Phi)$ is computed, and from this set we then obtain$Sat(\mathcal{P}(\mathsf{X}^I\Phi))$ by testing whether s satisfies the probability p calculated using Proposition 1.

The techniques used to model time bounded until $(\Phi \bigcup^I \Psi)$ are based upon the same techniques used to model check CTL[CES86], specifically using fixed-point characterisation. Here we let $I \ominus x$ denote $\{t - x | t \in I \wedge t \geq x\}$ and $\mathbf{T}(s, s', x)$ is the probability density of moving from s to s' within x units of time. $\mathbf{T}(s, s', x)$ is defined as

$$\mathbf{T}(s, s', x) = \mathbf{P}(s, s') \cdot E(s) \cdot e^{-E(s)x} = \mathbf{R}(s, s') \cdot e^{-E(s)\cdot x} \tag{9}$$

The residence time in state s at x is denoted by $E(s) \cdot e^{-E(s)x}$. Letting $\mathcal{I}$ be the set of all nonempty intervals $I \subseteq \mathbb{R}_{\geq 0}$ leads us to the following theorem.

Theorem 1 *Let $s \in S$, interval $I \subset \mathbb{R}_{\geq 0}$ with $a = \inf I$, $b = \sup I$ and Φ, Ψ be valid CSL formulae. The function $S \times \mathcal{I} \to [0, 1], (s, I) \mapsto Prob(s, \Phi \bigcup^I \Psi$ is the least fixed point of the higher order operator*

$$\Omega : (S \times \mathcal{I} \to [0, 1]) \to (S \times \mathcal{I} \to [0, 1]) \tag{10}$$

where

$$\Omega(F)(s, I) = \begin{cases} 1 & \text{if } s \models \neg\Phi \wedge \Psi \text{ and } a = 0 \\ \int_o^b \sum_{s' \in S} \mathbf{T}(s, s', x) \cdot F(s', I \ominus x) dx & \text{if } s \models \Phi \wedge \neg\Psi \\ e^{-E(s)\cdot a} + \int_o^a \sum_{s' \in S} \mathbf{T}(s, s', x) \cdot F(s', I \ominus x) dx & \text{if } s \models \Phi \wedge \Psi \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

A detailed proof of this theorem is found in [BHHK03] but we will informally cover the justifications for cases 2 and 3 here.

If s satisfies $\Phi \wedge \neg\Psi$ (as in case 2), the probability of reaching a state which satisfies Ψ from s within interval I equals the probability of reaching a direct successor, s' of s in x time units, multiplied by the probability of reaching a Ψ-state from s' in the remaining time interval $I \ominus x$.

For the case when $s \models \Phi \wedge \Psi$, the path formula φ is satisfied if no outgoing transition from s is taken for at least $\inf I$ time units. If state s is left before $\inf I$, the probability is calculated similarly to case 2. Following from Theorem 1 and Proposition 1 we can arrive at the following

Corollary 1 *For $s \in S$, and Φ, Ψ valid CSL state formulae*

1. $Prob(s, \mathsf{X}\Phi) = \sum_{s' \models \Phi} \mathbf{P}(s, s')$
2. *The function $S \to [0,1]$, $s \mapsto Prob(s, \Phi \bigcup \Psi$ is the lowest fixed point of the higher order operator $\Theta : (S \to [0,1]) \to (S \to [0,1])$ where*

$$\Theta(F)(s) = \begin{cases} 1 & \text{if } s \models \Psi \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot F(s') & \text{if } s \models \Phi \wedge \neg\Psi \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

The proof of this follows directly from Proposition 1, Theorem 1 as well as the property that $\mathsf{X} = \mathsf{X}^{[0,\infty]}$ and $\bigcup = \bigcup^{[0,\infty]}$

2.5.4 Numerical Analysis of CSL

From Corollary 1, the results we obtain are identical to the discrete time probabilistic case [Bai99] and as such the model checking of formulas using of the form $\mathcal{P}_{\trianglelefteq p}(\mathsf{X}\Phi)$ and $\mathcal{P}_{\trianglelefteq p}(\Phi \bigcup \Psi)$ can be done using familiar methods.

The vector $\underline{Prob}(\mathsf{X}\Phi)$ can be found by simple matrix calculation $\mathbf{P} \cdot \underline{i}_\Phi$ with $\underline{i}_\Phi$ being a vector characterising the set $Sat(\Phi)$ such that $i_\Phi(s) = 1$ if $s \models \Phi$ and zero otherwise.

For the unbounded until formula, $\underline{Prob}(\Phi \bigcup \Psi)$ will be the least solution of the following system of linear equations

$$\underline{x} = \hat{\mathbf{P}} \cdot \underline{x} + \underline{i}_\Psi \text{ where } \hat{\mathbf{P}}(s, s') = \mathbf{P}(s, s') \text{ if } s \models \Phi \wedge \neg\Psi \text{ and } 0 \text{ otherwise} \tag{13}$$

In general this system of equations has multiple solutions requiring the application of Gaussian elimination or an iterative method such as the Power method or Gauss Seidel in order to find the least solution.

Finding a solution to $Prob(s, \Phi \bigcup^{[0..t]} \Psi$ has proven to be more troublesome. Initial algorithms which attempted to solve the system of linear differential equations that can be reformulated from Theorem 1 prove to be unstable and computationally expensive. Using an iterative method to approximate the $Prob(s, \Phi \bigcup^{[0..t]} \Psi$ proved to have little advantage[HKMKS00].

2.5.5 Transient Analysis of $\bigcup^I$

Katoen, Hermanns et al have proposed an algorithm[BHHK03] that reduces model checking requirements containing $\bigcup^I$ to the problem of calculating transient probabilities on the CTMC, a well understood area of study with available efficient techniques.

First we note that transient state probabilities of a CTMC at time t correspond to calculating the probabilities of of path formula $\diamond^{[t,t]}at_{s'}$ with initial state s.

$$\pi^{\mathcal{M}}(s, s', t) = Prob^{\mathcal{M}}(s, \diamond^{[t,t]}at_{s'}) \tag{14}$$

Generalising this for arbitrary state formula leads us to

$$\pi^{\mathcal{M}}(s, Sat(\Phi), t) = Prob^{\mathcal{M}}(s, \diamond^{[t,t]}\Phi) = \sum_{s' \models \Phi} \pi^{\mathcal{M}}(s, s', t) \tag{15}$$

We now look at the general bounded until formula, $Prob(s, \Phi \bigcup^I \Psi)$, noting that intervals of the form $[t, \infty)$ can be treated as a combination of unbounded and and bounded until formula

$$Prob(s, \Phi \overset{[t,\infty]}{\bigcup} \Psi) = \sum_{s \models \Phi} Prob(s, \Phi \overset{[t,t]}{\bigcup} at_{s'} \cdot Prob(s', \Phi \bigcup \Psi) \tag{16}$$

This is achieved by splitting the general case into 4 types of bounded until formulae and showing how each can be reduced to one of two simple cases. We introduce here transformations on a CTMC using a CSL state-formula as a parameter. Given a CTMC $\mathcal{M}$, we define $\mathcal{M}[\Phi]$ to be M with all states that satisfy Φ made absorbing. We will only concern ourselves with the first two cases as they are the most relevant pertaining to the types of measures we will use later in our case study in chapter 6. Full proofs for all four cases can be found in [BHHK03].

Case 1: $\bigcup^{[0..t]}$ for absorbing goal states

Consider the path formula $\psi = \Phi \bigcup^{[0,t]} \Psi$ with all Ψ states absorbing. ψ is invalid when any state satisfying $\neg\Phi \wedge \neg\Psi$ is reached and all states reached thereafter are irrelevant, so we switch transition from $\mathcal{M}$ to $\mathcal{M}[\neg\Phi \wedge \neg\Psi]$.

Ψ is also absorbing, that is $\mathcal{M} = \mathcal{M}[\Psi]$ and so if at time t we are in a Ψ state, then we know ψ holds and therefore

$$Prob^{\mathcal{M}}(s, \Phi \overset{[0,t]}{\bigcup} \Psi) = Prob^{\mathcal{M}[\neg\Phi\wedge\neg\Psi]}(s, \diamond^{[t,t]}\Psi) = \sum_{s'' \models \Psi} \pi^{\mathcal{M}[\neg\Phi\wedge\neg\Psi]}(s, s'', t) \tag{17}$$

and this can be computed in the same way as we would solve (15) using standard techniques.

Case 2: $\bigcup^{[0..t]}$

For this case consider $\psi = \Phi \bigcup^{[0..t]} \Psi$ with an arbitrary CTMC $\mathcal{M}$. If we reach a Ψ state before t, then we need not consider any further behaviour of the system and it is safe to make Ψ absorbing without affecting ψ, reducing $\mathcal{M}$ to $\mathcal{M}[\Psi]$. This reduces the problem to (17). Also $\mathcal{M}[\Psi][\neg\Phi \wedge \neg\Psi] = \mathcal{M}[\neg\Phi \vee \Psi]$ and so

$$Prob^{\mathcal{M}}(s, \Phi \bigcup^{0..t} \Psi) = Prob^{\mathcal{M}[\Psi]}(s, \Phi \bigcup^{[0..t]} \Psi) = \sum_{s'' \models \Psi} \pi^{\mathcal{M}[\neg\Phi \vee \Psi]}(s, s'', t) \tag{18}$$

2.5.6 Uniformisation

In order to efficiently determine these transient probabilities techniques based on *uniformisation*[BHHK03] are used. This transforms the CTMC into a corresponding *uniformised* discrete time Markov chain(DTMC), which characterises the CTMC at state transitions.

For a CTMC $\mathcal{M} = (S, \mathbf{R}, L)$ the uniformised DTMC $unif(\mathcal{M})$ is defined as $(S, \mathbf{P}, L)$ where P is defined by

$$\mathbf{P} = \mathbf{I} + \frac{\mathbf{Q}}{q} \text{ for } q \geq max\{E(s_i)\} \text{ and } \mathbf{Q} = \mathbf{R} - diag(\underline{E}) \tag{19}$$

Substituting $\mathbf{Q} = q \cdot (\mathbf{P} - \mathbf{I})$ in the Taylor-MacLaurin series results in

$$\underline{\pi}(\alpha, t) = \alpha \cdot \sum_{i=0}^{\infty} e^{-q \cdot t} \frac{(q \cdot t)^i}{i!} \mathbf{P}^i = \sum_{i=0}^{\infty} \gamma(i, qt) \cdot \underline{\pi}_i \tag{20}$$

where

$$\gamma(i, qt) = e^{-q \cdot t} \frac{(q \cdot t)^i}{i!} \tag{21}$$

is the i-th Poisson probability and $\underline{\pi}_i$ is the state probability vector after i epochs in the DTMC with $\mathbf{P}$ determined recursively by

$$\underline{\pi}_i = \underline{\pi}_{i-1} \cdot \mathbf{P} \text{ with } \underline{\pi}_0 = \alpha \tag{22}$$

These probabilities can be computed efficiently using the Fox-Glynn[FG88] algorithm. However large difference in the orders of magnitude of the transition rates can have an adverse effect on this efficiency which we investigate in Chapter 5.

2.6 Conclusion

The combination of Continuous Time Markov Chains and Continuous Stochastic Logic is a powerful formalism for the specification and verification of dependability properties. The semantics of CSL provide sufficient operations enabling clear and intuitive specification of requirements. With the addition of the Duality theorem, a new dimension is opened with the addition of rewards allowing performability to be reasoned using CRL but with little computational overhead.

The algorithms developed to model check CRL requirements have progressed greatly in the last five years allowing for the efficient and numerically stable calculation of solutions. These algorithms have been implemented in the ETMCC tool, which we will cover in Chapter 3, allowing for practical results to be attained.

There are however a number of factors and limitations to take into consideration when dealing with CRL and MRM and we will explore these in greater detail in Chapter 5.

Chapter 3

Model Checking Tools for Performability Logics

3.1 Introduction

In recent years a number of tools have been developed to enable the efficient model checking of complex systems. Model checking tools follow the standard hierarchy of performance evaluation tools as shown in Figure 7.

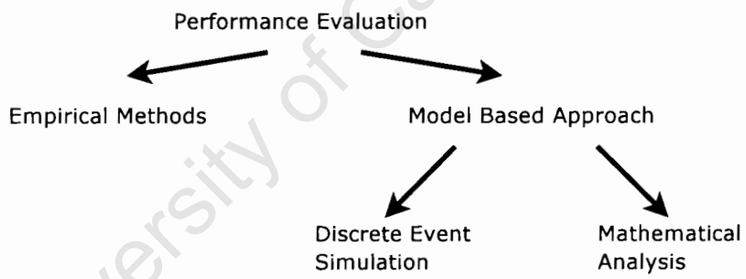

Figure 7: A hierarchy of performance evaluation techniques

Model checking falls under the model based approach, but tools based on both the discrete event simulation and mathematical analysis paradigms are available. We will primarily look at two tools: ProVer, a model checker based on simulation and ETMCC, which uses mathematical techniques in order to perform model checking.

Each of the two techniques has it's own advantages and disadvantages. Discrete event simulation is universal, and can be applied to any model type or formalism. However the computing time needed to obtain results from simulation can prove to be too excessive. Mathematical analysis can significantly cut down on the execution time needed by taking advantage of the mathematical and statistical properties of the model, but these properties may only apply in select cases.

3.2 ProVer

Developed at Carnegie-Mellon University, ProVer is a model checker which uses Monte Carlo simulation and statistical hypothesis testing to check whether requirements, expressed in CSL hold on a model. By using simulation, ProVer can solve a much larger class of problems, however it can not guarantee the validity of the results as other mathematical methods can. The authors of ProVer have however been able to guarantee that the error can be bounded[YS02]. We outline the method ProVer uses to do this in the next two sections.

3.2.1 Acceptance Sampling

Suppose s is a state in a Markov chain and Φ is a CSL expression that can be applied to that state. Let H_0 be hypothesis that Φ holds over s and H_1 the hypothesis that Φ does not satisfy s. We wish to guarantee that the probability of a false negative, where we accept H_1 when H_0 holds, is less than α while the probability of accepting a false positive is less than β.

Figure 8 shows a graphical representation of a possible sequential test used in acceptance sampling. A number of samples are taken and the number of positive samples out of the total samples is noted. Depending on which threshold the graph crosses the hypothesis is either accepted or rejected.

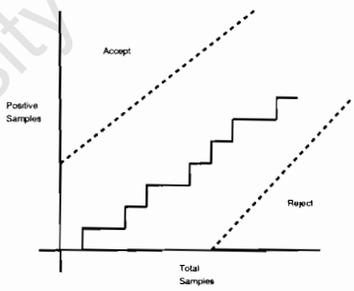

Figure 8: A simulation being tested using acceptance sampling

3.2.2 Restricting Error

To determine whether sampling should continue, we use a third hypothesis which represents indifference, that is we are unsure of whether H_0 or H_1 hold. If Φ is a CSL formula of the form $\mathcal{P}_{\geq\theta}(\Psi)$ that we wish to check against some state s. If p is the actual probability of Ψ holding over all paths starting from s then if $p \geq \theta$ then Φ holds in s.

We then wish to test $p \geq \theta$ against $p \leq \theta$ however in order to be able to freely set values for α and β we need to relax the hypothesis. We introduce an indifference region of $2 \cdot \lambda$, with $p \geq \theta + \lambda$ being H_0 and $p \leq \theta + \lambda$ being H_1. Let H_2 represent indifference and it will hold if p is within the indifference region. The following can then be guaranteed by an acceptance sampling test

$$Pr[H_0 \text{ holds}|\text{accept } H_1] \leq \alpha$$
$$Pr[H_1 \text{ holds} \vee H_2 \text{ holds}] \geq 1 - \alpha$$
$$Pr[H_1 \text{ holds}] \leq \beta$$
$$Pr[H_0 \text{ holds} \vee H_2 \text{ holds}] \geq 1 - \beta$$

If H_0 holds then Φ is true and if H_1 holds then it is false. However if H_2 holds we are still unsure of the truth value of Φ. However if we are in the area of indifference and p is close enough to θ then it does not matter whether it is above or below θ. If H_2 holds and we accepted H_0, then we can reason that Φ is true. Φ will be false if we accept H_1. From this we can then determine the desired error bounds to be

$$Pr[\Phi|\text{accept } H_1] \leq \alpha$$
$$Pr[\Phi|\text{accept } H_0] \leq \beta$$

Younes and Simmons extend the above strategy to handle cases where Ψ is composed of nested operators as well as covering both path and state formula of CSL. We refer the reader to their paper[YS02] for a more detailed explanation.

3.2.3 Implementation of ProVer

ProVer is implemented in C++, and at the moment is a text only application. Models are specified using a stochastic process[You03] definition language based on PDDL, a language used for deterministic planning, which allows for the state-space of the model to be generated at execution time, saving disk space in cases where the model size becomes unwieldly.

ProVer also has yet to fully support all operators in CSL and still lacks the ability to handle the $\mathcal{S}$-operator and unrestricted $\bigcup$ and X-operators.

Because there is no guaranteed upper bound for the number of samples that need to be taken to be able to verify a CSL requirement, Younes and Simmons suggest following Wald's[Wal45] proposal of setting an upper bound so that most error bounds are still respected. However from experience it seems that in most cases the number of tests needed for a simple CSL formula is relatively low. Only when we are checking a requirement of the form $\mathcal{P}_{\geq\theta}(\Phi \bigcup^{\leq t} \Psi)$ where both Φ and Ψ contain nested CSL expressions, do the numbers of samples become high, but requirements of this form are not typical.

In terms of performance Younes, Norman et al[YKNP04] have shown that statistical model checking techniques tend to be quicker than numerical tools when loose accuracy bounds are set. This allows them to be used quickly in the initial stages of the design when the accuracy requirements of the model are still quite small. As the need for accuracy grows in the finals stages of the design of the requirements of the model, tools based on numerical methods can then be used to determine the whether requirements are met with greater accuracy than the statistical based tools.

3.3 ETMCC - The Erlangen Twente Markov Chain Checker

Developed jointly by the Universities of Erlangen (Germany) and Twente (Netherlands), ETMCC[HKMKS99, HKMKS00] was one of the first model checkers developed for the purpose of model checking continuous time Markov chains. Initially developed to check CTMC using Continuous Stochastic Logic (Section 2.4) it has been extended to be able to use other variants such as aCSL[HKMKS01] and by us to handle Markov reward models and the corresponding Continuous Reward Logic which we detail in Section 3.3.3. We have discussed the various algorithms and techniques ETMCC uses in Chapter 2.

3.3.1 Tool Usage

On startup the user is presented with user interface in Figure 9. From here the user can either construct a model by loading the various component files of the model or load a previously defined project.

In it's default CSL run mode, ETMCC requires two files in order to construct a model, the transition (`.tra`) file and labelling (`.lab`) file. Currently the format of the `tra`-file is the same as the output generated by the TIPPTool[HM96], a process algebra specification tool developed at Erlangen. Labelling files are simple text files comprising a declaration of atomic propositions followed by a listing of each state and the atomic propositions which it satisfies.

Currently ETMCC can run in three modes: CSL, aCSL and CRL (implemented by us). As mentioned earlier the default mode is CSL but by including either a `-acsl` or `-crl` runtime flag on the

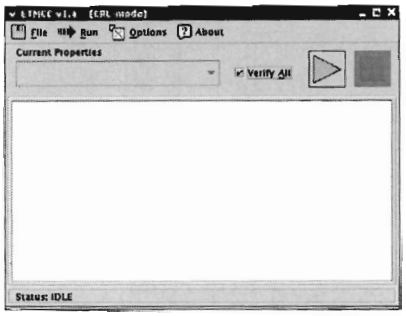

Figure 9: ETMCC on startup

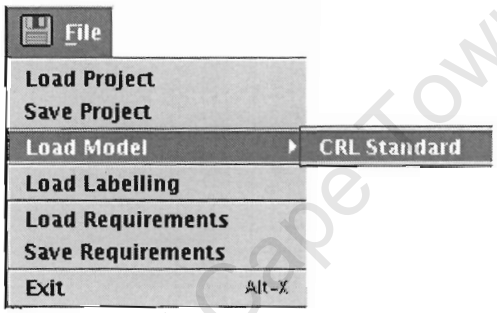

Figure 10: Loading of files including reward is supported

command line the tool will run in either aCSL or CRL mode respectively. In each of these modes various classes in ETMCC are replaced by classes written to handle the chosen formalism. In Figure 10 we see how the load file menu is changed when the CRL flag is set at run time.

Once the transition and labelling files are loaded, all that is left is for the user to specify the requirements on the model. This is done with the CSL property manager pictured in Figure 11. The property manager allows the user to easily specify syntactically correct requirement. Once the user has specified the requirements for the model the requirements can be tested and the results are output to log files.

3.3.2 Tool Structure

Figure 12 shows an overview of the underlying architecture of ETMCC.

The tool is divided into five separate sections. The core of the tool is the numerical engine which solve system of linear equations and Volterra integrals, dependent on the CSL requirement being checked.

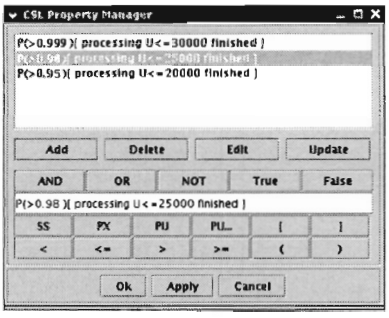

Figure 11: The CSL property manager

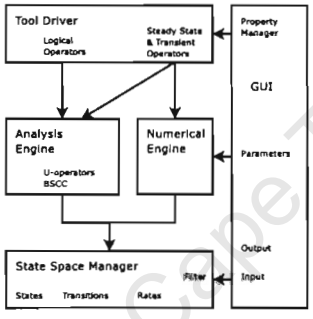

Figure 12: Overview of ETMCC structure

It is complemented by the analysis engine which uses graph theory techniques and algorithms to reduce the size of the state-space when dealing with CSL path formulae as well as computing the BSCC when steady state properties are checked.

The state space manager is responsible for the storage and manipulation of the state space and uses sparse matrices to combat potential state space storage size problems. The state-space also allows for different model types to be loaded by taking advantage of Java's dynamic class loading. The tool driver controls the model checking process, parsing the CSL requirements and calling the analysis and numerical engines where needed.

3.3.3 Implementing Reward Model Checking

Due to the nature of Duality(Section 2.4.2), implementation of the ability to handle rewards in ETMCC was relatively trivial. The original authors of ETMCC in their design and implementation of the tool made sure to allow for further extensions to the models the tool can handle.

In our case the changes needed to the tool were only needed in the IO subsystem of the tool. Our MRM generation tool, produces a simple file of the form

```
STATES 1001
TRANSITIONS 2998
1 0.250000
2 0.500000
3 0.750000
...
1000 250.000000
1001 0.000001
1 2 0.083333
1 1001 0.000342
2 1 0.000046
...
999 1001 0.000342
1000 999 0.022831
1000 1001 0.000342
1001 250 0.041667
```

The file begins with two declarations defining the number of states and transitions in the file. Immediately following that is a listing of each state and the reward assigned to that state. Following the reward declaration is the listing of the transitions, with the originating state given first, followed by the terminating state and then the transition rate.

We modified the already existing CSL model parser class, by extending it to first read in the reward declarations, storing them temporarily. Then when each transition is read in, the rate for that transition is divided by the reward associated with the originating state. This modified transition is then stored in the state space manager. The result is that ETMCC can now reason about rewards (but not the time) of a MRM.

This procedure required the modification/addition of only 2 classes in ETMCC out of the 162 classes that comprise the tool. CSL and CRL also share the same syntax for their requirements so we can use the CSL property manager as is to specify CRL requirements further cutting down on the code needed to be able to handle MRM.

3.4 Other Tools

While the model checking of continuous time Markov chains is undoubtedly a new field a number of tools are under development at institutions around the globe.

3.4.1 PRISM

Developed at the University of Birmingham, PRISM[KNP02] is a model checker for CTMC as well as Markov decision processes. Developed in a mix of Java and C++, it allows models to be specified in a language based on the reactive modules formalism[AH99].

It uses the same general approach to the solving of CSL requirements as ETMCC and in many cases uses the same algorithms and techniques.

3.4.2 SMART

SMART[CM96] is a tool using both simulation and numerical techniques in order to solve a wide variety of problems. It uses a strongly typed language which allows the user to construct models that are equivalent to stochastic Petri nets, although new modelling formalisms such as queueing networks can be added to the tool.

SMART also has the advantage of being able to distribute it's processing amongst various workstations which is especially useful when simulation is being performed on a single model with many different parameters. It also allows for parallel state space generation.

While not originally designed to do model checking, later versions of SMART have added the ability to model check models using a logic expressed in Computation Tree Logic, which forms the core of CSL.

3.4.3 Performability Analysis Tools

Besides the specialised model checking tools listed a number of other tools are also available for performability analysis. Haverkort and Niemengeers[HN96] have surveyed a number of these tools which have been developed in the last two decades. At the time the paper was written (1996) they noted a shift from low level formalisms, such as plain Markov reward models, to higher level formalisms such as Markov reward models, Petri Nets and Queueing Networks, which we will discuss in Chapter 4. This has continued with the emergence of model checkers where requirements can now form part of the model providing additional expressive power.

3.5 Conclusion

We have presented a number of tools that are able to model check CSL requirements against continuous time Markov chains. The advantages of each technique, whether it is the ease of finding a solution using simulation or the concrete answers attainable from mathematical analysis, must be weighed carefully against the disadvantages.

ETMCC, while still being a prototype, is quite mature and is easily extensible with a well documented interface and the ability to handle multiple input formats. For those investigating models comprising Markov chains and who wish to take full advantage of CSL operators ETMCC would be the preferred choice. We use ETMCC extensively in our case study performed in Chapter 6.

ProVer, while being a tool with much fewer features, still has some interesting application areas thanks to it's ability to model semi-Markov and generalised semi-Markov processes due to it's use of simulation. We also see an exciting combination of both numerical and event simulation techniques which we detail in Section 7.2.2.

Chapter 4

Specification Techniques of Markov Reward Models

4.1 Introduction

The usage of MRM as a technique for performability analysis has led to a large number of publications dedicated to the various solution techniques available, which we covered in Chapter 2. However there has been a lack of focus on the specification and modelling techniques of MRMs using higher level formalisms, mainly due to the fact that the various specification techniques used to model CTMC can be applied to MRM with little or no modification.

In this chapter we summarise the findings of Haverkort and Trivedi's exhaustive survey paper *Specification Techniques for Markov Reward Models*[HT93]. We pay special attention to Production Rule Systems and demonstrate the ease with which a usable system can be implemented by taking advantage of the features found in a suitable high level programming language to significantly reduce code size and implementation time.

4.2 Specification of Markov Reward Models

Haverkort and Trivedi put forward three requirements on the usage of specification techniques as applied to MRMs. These are

- User-oriented requirements
- Requirements with respect to tools

- Classification criteria for specification techniques

4.2.1 User-oriented requirements

User oriented requirements, as the name suggests, are requirements on a specification technique to enable the end user to specify a model with greater freedom and without an excess of complexity. The designer or engineer working on a model is primarily concerned with the solution to the problem presented, and should not be concerned with the techniques that were used to find that particular solution.

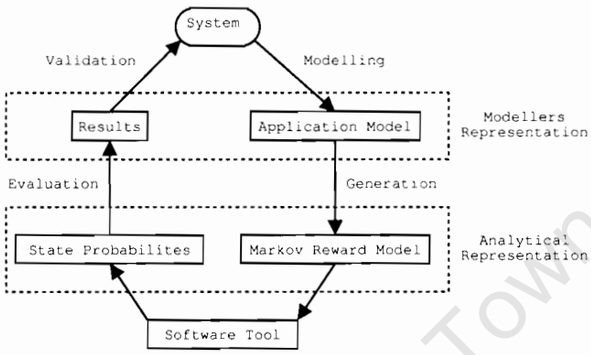

Figure 13: The analytical and modellers representation of a system model

The system designer need not even be aware of the underlying model. This underlying model, called the *analytical representation*[BdSM87], is a representation of the problem that can be directly solved via the various numerical techniques. The representation of the model used by the designer is the *modellers representation*. The relationship between these two models is shown in Figure 13. From Figure 13 the cyclical feedback nature of the relationship, common in a variety modelling techniques, is quite clear.

The modellers representation should also encompass the measures of interest. In our usage these measures are expressed in CSL which while being separate from the model representation of the system are still an integral part of the performability analysis process.

4.2.2 Tool Requirements

Any software tool to generate and solve MRMs will by definition be complex. Therefore a modular approach to the design of such tools is highly favourable. This approach allows the tool to be

open ended, to easily support new data formats and to extend itself to take advantage of any new solution techniques. While in the past tools were built for efficiency using compiled languages such as C/C++, with the rise of relatively cheap value of powerful processors (thanks to Moore's Law), these days it is much more beneficial to write tools in higher level languages.

The tools we discussed in Section 3.3 such as PRISM[KNP02] and ETMCC[HKMKS99] are now written in languages such as Sun Microsystems' Java. While these languages carry a computational overhead, their modular, object oriented nature and extensibility provide a large advantage. The speed afforded by processors today makes the processing overhead of these languages almost negligible.

A complex specification should also be able to be broken up and each component entered separately in to the tool. The CTMC, labellings, performance measures and other components should be as independent as possible to allow for as much model reuse as possible.

4.2.3 Classification Criteria

We list the following criteria for classifying specification techniques for MRM

Domain-orientedness: Is the specification technique valid in only one particular domain? Is it based on specific rather than general assumptions?

Class of measures that can be supported: Certain specification techniques might not be able to support the measures we are interested in.

Modelling freedom: Does the specification technique force the designer to construct structured models due to a number of constraints?

Structuredness: Does the specification technique allow the designer to construct "free form" models or are models restricted to a hierarchical structure?

Abstraction from underlying mathematical model: To what extent is the analytical model hidden from the designer? Is knowledge about the analytical model and the techniques used to solve it required to model the system accurately?

Degree of completeness: How well does the specification technique support all possible MRMs?

4.3 Specification Techniques

4.3.1 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) were originally developed for the performance and correctness analysis of systems but have found increasing usage in other areas.

SPNs consist of three components, a set of places P, a set of transitions T and a set of arcs A. Places are not connected directly to other places but are first connected to a transition via an arc. Each place contains one or more tokens, which can increase or decrease over time. The distribution of tokens over the places is called a *marking*, with each distinct marking analogous to a state in a CTMC.

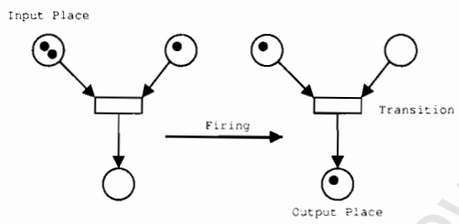

Figure 14: Firing of a transition

A place which has an outgoing arc to a transition is said to be a input place of that transition, while a place that has an arc incoming from a transition is an output place. When all of a transitions input places contain at least one token, that transition becomes enabled and may be fired. Firing of a transition removes one token from each input place and produces a token in each output place. Figure 14 shows this firing, after which the transition becomes disabled.

The rate of firing of the transition is usually assumed to be an exponentially distributed, and with the initial marking of an SPN known, all subsequent markings can be computed as long as the number of tokens in each place is bounded. When more than one transition is enabled, a *race condition* is reached with each transition firing according to a exponentially distributed random variable.

The rewards can then be obtained as a function of the markings of the SPN. The final SPN and reward structure is referred to as a Stochastic Reward Net[MCT94, BCC+92] which is a useful formalism for specifying reliability measures. Finally the labellings for the Markov chain are derived from the labellings on the places in the SPN. If a token is present in a place then the labelling on that place is also assigned to all states in the underlying Markov chain whose markings include the token.

In terms of structuredness, SPN are extremely free form, allowing the user to construct models that are not as limited as other formalisms, such as queueing networks. However this can lead to extremely complicated diagrams which obscure the model. This has lead to a number of extensions being added to SPNs, such as colored Petri nets and inhibitor arcs[BK95], which simplify the modelling process significantly. The unstructured nature of SPN also affects the domain orientedness, allowing them to be applied to a wide range of problems in many problem domains as well as giving them a high level of completeness.

For the case study we will perform in Chapter 6, we are looking for a formalism that is more structured than SPN, that can easily be modified especially when dealing with large models. When dealing with large complex SPN models, the effort needed to change large sections of the model can be quite tedious. Researchers have even begun to construct mappings between SPN and other higher level formalisms[BDM02], in order to reduce the disadvantages of using SPN while still taking advantage of it's advantageous correctness verification properties.

4.3.2 Queueing Networks

Queueing networks is a formalism that is often used to model and analyse communicating systems. They are quite intuitive in use, and can represent certain systems much more closely than other modelling formalisms. They are well suited when dealing with capacity planning and service time analysis[FF97].

QNs are comprised of two resources, queues where jobs (or customers) are accumulate and servers where they are processed. Servers are dependent on jobs being available and as long as a queue has jobs available those jobs will eventually be processed. Jobs can have differing classes which can affect the order in which they are processed as queues need not be purely FIFO but can be sorted on priority. Servers however will process jobs on a first-come-first-served basis. Figure 15 shows a simple queueing network involving 2 servers removing jobs off a queue and processing them.

If we set each queue to be a finite size and each servers service time to an exponentially distributed distribution then we can extract from the QN a Markov chain which models the behaviour of the queue. A state is represented by a vector with each entry in being another vector listing the different job classes at each queue. This is similar to the equivalence between markings of an SPN and states in the underlying Markov chain. Figure 16 shows the Markov chain extracted from the QN specified in Figure 15 with the coordinate in each state representing the number of jobs in each queue.

From Figure 15 the highly structured nature of the Markov chains extracted from QNs is apparent. Unlike SPNs, the modelling freedom of QNs is quite limited with the user having little ability to place exceptions to certain states being generated or restrict transitions between states. If we wish to focus on the dependability of a system where components fail during the course of operation it

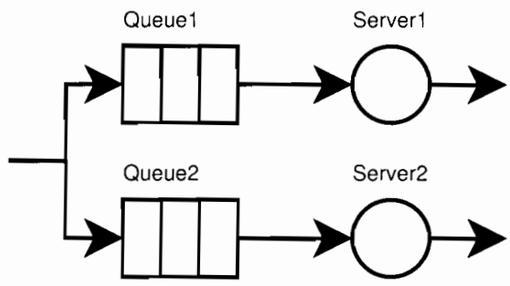

Figure 15: A sample queueing network

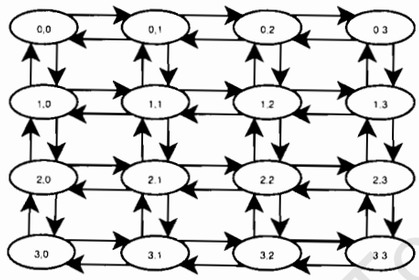

Figure 16: A Markov chain derived from Figure 15

would be extremely difficult to model with QNs and while being possible, would produce a model that is not very intuitive or representative of the system at first glance.

When assigning state labellings QNs do have some apparent advantages as there is an almost inherent abstraction of state labellings in the QN. If we look at Figure 15 we initially have two apparent labellings, *Queue1* and *Queue2*. However each of these needs to be divided further to represent the number of states in each queue, so *Queue1* would now be split *Queue1-1*,*Queue1-2* and *Queue1-3*. However we do not lose any abstraction by removing *Queue1* and *Queue2* as we have also defined labellings on the server. When *Server1* is satisfied then all the labellings on *Queue1* are satisfied as well. We find then that it is quite straightforward to extract labellings from the initial QN specification, with labellings on the queue applying to single states and labellings on the server providing labellings that serve as an abstraction over a number of states.

To complete the specification of a MRM from a QN we will also need to assign a reward structure to the QN. One apparent advantage of using QNs to model MRM with is the variety of reward structures we can assign to a single model. There are a number of properties available at all possible states of the model that are suitable to be considered as reward measures. A reward structure can

be constructed out of any combination of measures observing the number of servers active, number of full or empty queues, number of jobs of a certain class in a specific queue, number of jobs in all queues and so on.

In terms of the criteria laid out by Haverkort and Trivedi, QNs are quite domain oriented and restricted in flexibility for modelling. However they provide a significant degree of abstraction from the underlying Markov reward model and for certain problem domains are more suitable than other formalisms.

While the QN formalism seems better suited to our interests than SPN, the strict limitations and domain orientedness lead us to conclude that for the purposes of modelling the performance requirements of a system we are interested in, that they are not suitable.

4.3.3 Production Rule Systems

Of all the formalisms for specifying MRM, production rule systems (PRS) are the most general, having no restrictions on the type of measure which they can model. This modelling freedom, while seemingly advantageous at first can lead to problems when modelling which we identify in this section and section 4.4.

A PRS is a formalism based on placing a number of conditions on states and then checking whether or not these conditions hold. The conditions specified then generate suitable transition and reward rates. The following sample production rule

```
if satisfied(CONDITION,STATE) then
        NEWSTATE = reachable(STATE,CONDITION)
        RATE = lambda(STATE,NEWSTATE)
        addToModel(STATE,NEWSTATE,RATE)
```

illustrates the components of a PRS governing the behaviour of transitions in the model. When a suitable condition is satisfied by the state, we generate the state (or states) reachable from under that particular condition. We also generate the transition rates that apply between the state and all states reachable from it under this condition. This process is then repeated for all the new states generated. A similar production rule would also be applied to the state when generating the reward.

Production rule systems can suffer from a lack of modelling clarity. When modelling with SPN or QN the structure of the underlying Markov chain extracted from it is directly related to the structure of the QN/SPN. However due to the nature of constructing PRS, this relationship is not always apparent. Conflicting rules, where multiple transition rates are assigned between two states or multiple rewards assigned to a state, can occur as the complexity and number of rules used in the model increase.

In Section 4.4 we describe the implementation of a simple PRS. We designed this system with the case study in Chapter 6 in mind as a way to quickly specify highly uniform performability models.

4.3.4 Other formalisms

Besides the formalisms listed by Haverkort and Trivedi there are a number of other techniques that can be used to describe a MRM. Thanks to the Duality Principle (2.4.2) and the transformation allowed between MRM and CTMC, any formalism that can describe a CTMC can then be used to describe a MRM with little modification. More sophisticated formalisms such as process algebras[HK00] are now made available for the specification of MRM.

4.4 Implementation of a Production Rule System using High Level Languages

4.4.1 Feature of High Level Languages

In recent years there has been a marked rise in the use of high level languages , also referred to as 4th generation languages. These languages are usually interpreted and are. The most popular of these languages is undoubtedly Perl with a number of other languages enjoying various levels of support. We have selected the Ruby[TH00] programming language for this section. Ruby is an object oriented language scripted language developed since the 1991 by Yukihiro Matsumoto. It features a complete OO model which allows even basic data types such as integers to be treated as objects.

Interpreted languages have a number of advantages over compiled languages, especially in the area of dynamic evaluation of code. In Ruby any string can be inserted into execution of the program merely by calling the `eval` function. The following code sample

```
code = "a = 2;print(a)"
a = 3
eval(code)
```

will evaluate the contents of the string stored in `code` and change the value of `a` from 3 to 2 and output it to screen. The instructions stored in `code` can be any combination of Ruby expressions.

We selected Ruby to write our production rule system due to it's strong object model, it's powerful reflection features and it's ability to dynamically generate and run code.

4.4.2 System Overview

The `mrmGen` tool we produced is comprised of a number of *generators*, with each generator concerned with a specific component of the MRM, either a state labelling, reward or transition. Each of these generators inherits from a *conditional generator* which defines the interface each generator must implement. Figure 17 shows the various classes that make up mrmGen.

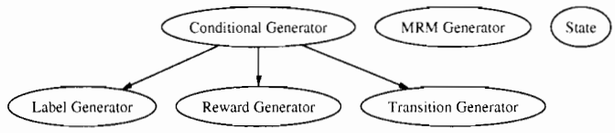

Figure 17: Overview of the class structure of mrmGen

Besides the specialised generators, the tool comprises a controller class, `MRM Generator`, and a class that represents each state, `State`.

The user defines the various generators using a set of textual delimiters, which indicate to the tool whether the code within the delimiter is one of the various generators, a state space definition or a label declaration. We explain these various declarations in Sections 4.4.4 – 4.4.6 in greater detail.

To generate a labelled Markov reward model, the user specifies the various generators and declarations in an input file with an extension of `.gen`. The generator is run on this file and three output files are produced. Assuming our input is `model.gen`, the tool will produce a file containing the state labellings (`model.lab`), transitions (`model.tra`) and rewards (`model.rwd`). The tool was written to produce output compatible with ETMCC and to this end `model.tra` and `model.rwd` are merged to produce `model.mrm`. Figure 18 shows this flow of files from mrmGen to ETMCC. The `model.req` file is generated internally by ETMCC and holds the requirements expressed in CSL but we include it for completeness.

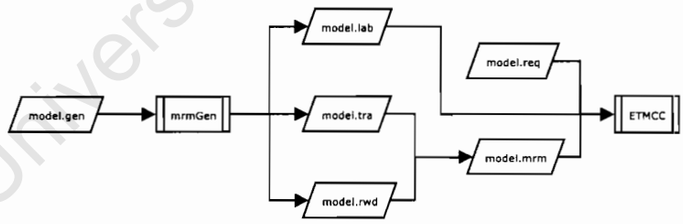

Figure 18: Interaction of files generated by mrmGen and used by ETMCC

4.4.3 Syntax

Because we are using the Ruby interpreter to check conditions and rewards the syntax of the the declarations is merely that of the Ruby language itself. However certain declarations require compliance to certain operators or data types while others are much less restrictive.

In Ruby every expression returns a value. As an example the expression

```
a = if condition
        "I am a value"
    end
```

will return a string object which is set to `"I am a value"` and assign it to variable `a` if the condition evaluates to be true. We exploit this in our tool by allowing the user to define the right hand side of the assignment and then using the dynamic nature of code execution found in Ruby to evaluate the code and capture the result.

When a condition has to return a value the last statement executed should be the return value. If no value is returned then a *nil* (essentially equivalent to a null pointer) value is returned. In the following expression

```
b = (if condition
       "I am a value"
     end
     "I am another value")
```

no matter what value of the condition the variable `b` will always have the value `"I am another value"` as that string is the last statement on the right hand side of the expression and so it's value will always be returned.

4.4.4 Declaring State Space

When using mrmGen we declare our state space explicitly beforehand. The following full state space declaration

```
dimension:
[[(1..10),(1..10)]]
:dimension
```

will produce a state space containing 100 states, with each state being referenced by a co-ordinate of the form (x, y) where x and y both range from 1–10.

The state space is defined using an array with each element in the array either defining a range of states or a singular state. If we extend the above example to `[[(1..10),(1..10)],[0,0],[200,200]]` then the state space generated will have an additional two state in addition to the 100 states previously generated.

The state space declaration also takes the role of the state variables. Each number in the state coordinate being equivalent to the value of a state variable. In the example given above we have two possible state variables.

Besides having a user defined coordinate, each state is also assigned a unique state number. In the extended example given in the above paragraph the state with co-ordinate (1,3) will have state number of 3, (200,200) will have a number of 102.

4.4.5 Reward Generation

Generation of rewards is done by placing conditions on the coordinate of the state. The declaration

```
reward:
if coord[0] < 5
        coord[0] * 5
else
        coord[0] * 10
end
:reward
```

will assign to each state a reward that is a multiple of either 5 or 10 depending on the value of it's 1st coordinate. If there are multiple definitions for a reward in a specification of a MRM, the reward that was declared last will be the one used for that particular state.

There are little restrictions on a reward generator and the entire Ruby syntax and built in functionality is allowed when in this declaration. However to be of any use the declaration must return either a number or `nil`. The only variable made available to declaration initially is `coord`, which is an array holding the coordinates of that state.

4.4.6 Label Generation

Before specifying labels for the model, we define the labels that are allowed in the model. This enables us to check whether any illegal labels are specified in the model, as well as providing error

checking. The following declaration

```
labeldeclaration:
Failure,Processing,Critical
:labeldeclaration
```

defines the usage of three labels used in the model. Once these labels are defined we can then define when a state will have one or more particular labels attached to it. This declaration is strictly a comma separated list of strings incorporating no Ruby syntax or functionality.

Once the labels are previously declared we can then construct conditions on which labels apply to states. Due to the possibility of states having multiple labels, we allow multiple definitions for a state. The following two label definitions

```
label:
if reward == 0
        "Failure"
else
        "Processing"
end
:label

label:
if coord[0] <= 5
        "Critical"
end
:label
```

will both be applied to states. All states will either have a *Failure* or *Processing* and those whose first coordinate is less than five will also have *Critical* as a state labelling.

As with the reward condition the contents of the label condition can be any number of valid Ruby statements, and in addition to the `coord` of the state, the `reward` is available as well. Each label condition should return a valid string or a nil to be useful.

4.4.7 Transition Generation

Unlike the other generators, to generate a transition we have to place conditions on two states, the originating and destination state. These are specified in the declaration as `from` and `to` respectively. The declaration

```
transition:
if (from[0] - 1 == to[0]) and (from[1] == to[1])
        0.03
end
:transition
```

places a transition with rate 0.03 between two states that have coordinates $(x-1, y)$ and (x, y). As with the reward generator if there are multiple definitions for a transitions between two states, the last transition to be generated is used. Two variables are initially made available to the condition, `from` is an array containing the coordinates of the state where the transition originates and `to` is the corresponding array for the terminating state.

4.4.8 Performance

The performance of the tool was not a primary factor during implementation, rather we were more interested in developing a usable production rule system in as little code as possible in order to highlight the usage of a high level language. However the usage of high level languages does add the execution speed as high level commands are translated into native commands and executed, which can lead to an almost 75% increase execution time for certain applications[Bag].

Performance degradation in the tool is due mainly to the manner in which the tool applies the generator that controls the transitions to the states. For every state defined we iterate through every other state checking the various transition conditions against both states. This could lead to a worst case performance of $\mathcal{O}(n^2)$. Figure 19 shows the performance of the tool as it generates a simple linear MRM of increasing size.

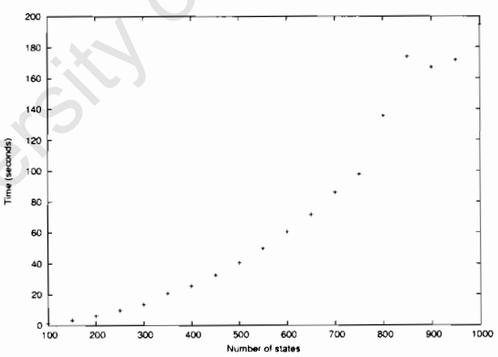

Figure 19: Performance of the mrmGen tool as the model size is increased

The performance is acceptable when dealing with hundreds of states but begins to degrade when approaching a state space numbering thousands.

When computing the transitions it is possible to reduce the execution time by precomputing destination states instead of iterating over the entire states-pace for every state checking whether the transition condition holds. This is possible if the user specifies beforehand explicitly which states relative to the current state should have transitions. This should enable us to reduce the execution time to $\mathcal{O}(n)$ in the future. As a proposed example the following declaration

```
transition:
  terminate:
    coord[0]+1,coord[1]+1
  :terminate

  rate:
    reward * 0.75
  :rate
:transtion
```

would be executed by first finding the coordinate of the terminating state, in this case $(x+1, y+1)$ relative to the originating state, determining the reward from the specification, calculating the state numbers of both the originating and terminating states and then outputting these state numbers along with the rate to file. Using the dynamic nature of Ruby this can be easily implemented.

In cases where the state space is extremely large and where full state space exploration is required there are a number of parallel generation techniques[CGN95, KHMK00] which can be implemented. Again the usage of Ruby would prove advantageous as numerous distributed computing libraries are available[TH00].

4.5 Conclusion

The specification formalism used when constructing a model is often not a crucial factor. However deciding on the correct formalism to use is important and can affect the ease with which our system is modelled. The variance in domain independence, rigidity of structure and abstraction of each differing formalism means that a suitable formalism should be chosen on a case-by-case basis.

We have demonstrated in the case of production rule systems the advantages that are gained by writing a production rule tool in a suitable high level language. This approach allows us to use the interpreter of the HLL to execute the conditions of the various production rules. The simple syntax

of the HLL allows the user to easily specify rules while still retaining a significant degree of control and flexibility.

While the performance of our HLL system is affected by larger state spaces, for specifying system models required for the case study performed in chapter 6, it is adequate. The nature of the case study we perform in Chapter 6 would seem to indicate the usage of either a PRS or QN as the specification technique. While the PRS we produce is quite similar to QN in the nature of MRM it can produce, our system is still more dynamic in that we can arbitrarily restrict and create transitions and states according to a set of rules and so it is this that we choose to use in our case study.

Chapter 5

Limitations of Model Checking Markov Reward Models

5.1 Introduction

As we indicated in Chapter 2, the algorithms and logics used in the verification of CRL properties still have limitations inherent in their use. In this chapter we will look primarily at limitations dealing with zero reward states, a limitation present on MRM, as well as on the actual logic of CSL/CRL and the available algorithms currently in use.

We also propose a number of solutions where possible and highlight areas where solutions are not possible or where they are possible but so far have not been implemented.

5.2 Zero Rate Reward

5.2.1 Restrictions

Due to the Duality Theorem (Section 2.4.2), we are restricted to only being able to use positive non-zero real numbers as rewards when solving MRM using analytical tools such as ETMCC as due to the transformation that is applied to the CTMC, possible divisions by zero might occur. This restriction, while seemingly not significant, prevents us from modelling a system which resides in an idle state for some period of time where zero reward is earned while in that state. Systems with idle states include such diverse cases as a server which have no jobs to process or a mobile device in

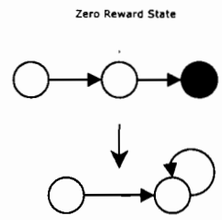

Figure 20: Transformation of Markov reward model using absorption to remove zero reward states

sleep (or power-saver) mode. In these states the systems earn no reward or a small enough reward to be negligible.

Clearly before verification can proceed we need to eliminate zero reward states from the MRM. However we need to guarantee that neither the numerical or logical properties of the MRM are altered in any way.

5.2.2 Removal of Zero Reward States

Beaudry[Bea78] proposed a simple algorithm which removes zero reward states by making all such states absorbing which effectively removes the state from the model. However it has severe limitations in that once a zero reward state is reached, the verification process is halted. Models which have a large number of zero reward states, or which encounter zero reward states relatively early will be significantly hampered by this constraint. With models with this characteristic we risk removing large parts of the model connected to the zero reward state by making that state absorbing. This is particularly relevant when model checking with CRL when dealing with path formula. If a zero reward state satisfies the LHS of a $\bigcup$-operator, using Beaudry's method the resultant transformed Markov chain will no longer be able to reach a state that satisfies the RHS of the operator.

Figure 20 shows this graphically. In the top figure we have a simple path through Markov chain with a zero reward state that satisfies the *white* atomic proposition. Transforming the chain according to Beaudry's method makes the state absorbing and prevents the *black* state from being reached.

Ciardo et al.[CMST90] expanded on the work of Beaudry by defining a new MRM where the zero reward states are removed from the MRM and replaced with a probabilistic switch without changing the behaviour of the MRM. Knottenbelt[Kno96] also proposes an "on-the-fly" algorithm of eliminating these *vanishing states* during state space exploration.

Wholesale removal of zero reward states however can be dangerous in model checking as we run the risk of modifying the logical property of the MRM.

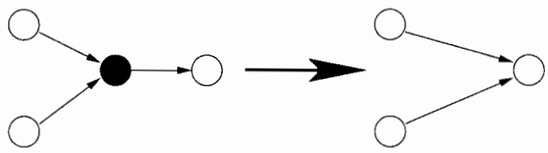

Figure 21: Removal of a zero reward state and loss of logical property from a simple Markov chain.

Figure 21 shows a simple Markov chain with states which satisfy either of the two atomic propositions, *black* or *white*. If the *black* state has zero reward and is removed, it is possible that a CRL statement that was valid previously (for instance any CRL statement using $white \bigcup black$) is no longer valid in the transformed chain as *black* no longer exists and as such the absorbing state will never be reached.

There is also the problem of entering a *timeless trap* which is found when a cycle of vanishing states forms a strongly connected component of the Markov chain. These traps render performance analysis impossible.

5.2.3 A Refinement to the Algorithm

In the process of model checking the requirements, expressed as statements in CRL, are explicitly known in advance. By examining the requirements we can decide which zero reward states can safely be removed without affecting the logical properties of the MRM.

We use a subset of CRL operators comprising the path$(\mathsf{X}, \bigcup)$ and state $(\neg, \wedge, \vee)$ formula. We can ignore the probabilistic operators $(\mathcal{P}, \mathcal{S})$ operators for the time being and still be guaranteed to remove all potential zero reward states as $Sat(\mathcal{P}(\psi)) \subset Sat(\psi)$ where ψ is a path formula.

CRL formula can easily be stripped of the probabilistic operators while leaving the path and state formula, for instance the CRL statement $\mathcal{P}_{\trianglelefteq p}(\Phi \bigcup \Psi)$ will become $(\Phi \bigcup \Psi)$. Simplifying the statement in this way does not remove any relevant information about the structure of the expression and so we are still able to reason about the dependencies of the terms in the expression.

To examine the CRL requirement, we insert the statement into an expression tree via means of a stack. Figure 22 shows the sample expression tree of a sample CRL statement. From the expression tree we can calculate if it is safe to remove a state that satisfies a certain atomic proposition or sub-expression.

If we look at Figure 21 and apply a CRL expression of the form $\mathcal{P}_{\geq 0.95}(white \bigcup black)$ then by first making sure that all *black* states are made absorbing (due to it being found on the RHS of a $\bigcup$-operator) before zero reward states are removed we can prevent the loss of any logical properties.

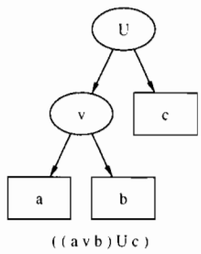

Figure 22: A simple CRL expression inserted into an expression tree

Of the three state operators, $\vee$ (And), $\wedge$ (Or) and $\neg$ (Not). The $\vee$- and $\wedge$-operators are binary while the $\neg$-operator is unary. We decide when to remove states or make them absorbing depending on which operators are being used in the CRL expression as follows.

And Operator

A CRL expression using the $\wedge$-operator is denoted as

$$\Phi \wedge \Psi \tag{23}$$

with Φ and Ψ being path formula. When dealing with this CRL operator (and in the case of the $\vee$ and $\neg$ operator) we use the semantics of sets and logic. For this particular operator we would find the intersection of the set of states that satisfy Φ and the set that satisfies Ψ.

In terms of our expression tree we would find the states that we can remove from the left subtree of the expression and intersect that with states that satisfy the same conditions on the right subtree. This is performed in a recursive manner on the tree, with the states that can be removed from Φ and Ψ determined independently of any other expression.

Or Operator

Expressed as

$$\Phi \vee \Psi \tag{24}$$

we can determine which states are safe to be removed from this expression in a similar manner as for the $\wedge$-operator, however in this case we union the set of states that satisfy Φ and Ψ instead of intersecting them.

Not Operator

The $\neg$-operator is quite simple and is expressed as

$$\neg\Phi \tag{25}$$

Unlike the $\wedge$- and $\vee$-operators, we can not determine the states that can be removed immediately from the model when solving the the $\neg$-operator. In this case the states that can be marked for safe removal are those that do not satisfy Φ. Using the labelling function of the MRM one can find the states that do satisfy Φ and from that determine the set of states that do not satisfy the expression and therefore can be removed if needed.

The path operators consist of two operators, the $\bigcup$-operator (Until) and the X operator, of which the $\bigcup$-operator is binary and the X operator is unary in nature.

Until Operator

For the simplest case

$$(a \bigcup b) \tag{26}$$

we merely have to inspect states that satisfy atomic proposition a. If the state has a reward of zero we can remove it without affecting the logical property of the model.

In more complex cases we can have nested state formula on the left or right hand side of the the until operator. For the left hand case

$$(\overbrace{(a \bigcup b)}^{\Phi} \bigcup c) \tag{27}$$

we begin with the nested statement Φ. We handle the nested CRL statement in the same manner as in (26). We can then assign to each state that satisfies the nested state formula Φ a labelling, d, which reduces (27) to an equation of the form $(d \bigcup c)$, as in (26).

However we do not repeat the steps taken to remove the zero reward states as we did in (26) due to the fact that $Sat(d) \subset Sat(a)$ and we had previously removed all zero reward states that satisfy a.

In the case where the nested state formula is more complex of the form:

$$((\Phi \vee \Psi) \bigcup \Upsilon) \tag{28}$$

where Φ and Ψ are sub-expressions separated by some logical state operator (in this case the *or* operator), then we merely follow the case as for (27)

For cases where the nested path formulae fall on the right hand side of the until operator, as in state formula of the form

$$(a \bigcup \overbrace{(b \bigcup c)}^{\Phi}) \tag{29}$$

we need to ensure that we do not remove any states satisfying b, even though it is the LHS term of a $\bigcup$-operator it still forms part of a state formula that is on the RHS of a parent $\bigcup$-operator. Some complication can occur where we have expressions that are satisfied on both sides of an $\bigcup$-operator such as

$$((\Phi \bigcup \Xi) \bigcup (\Psi \bigcup \Phi)) \tag{30}$$

In this case we have a sub-expression, Φ, that is on both sides of the root $\bigcup$ operator. We can not remove all Φ-states without analysis of the RHS of the root $\bigcup$-operator as doing so could cause Φ states that satisfy $\Psi \bigcup \Phi$ to be removed and CRL statements of the form in (30) will no longer be valid.

Next Operator

The next operator is used in CRL statements of the form

$$\mathsf{X}\Phi \tag{31}$$

The operator is restrictive in that it is a stopping condition, much like the formula on the RHS of a $\bigcup$-operator or the $\neg$-operator. As with the $\neg$-operator we can not immediately determine the states that are suitable for removal from an expression in the form of (31). We can determine the set of states that can reach a Φ-state within one transition, and therefore the states which are safe for removal, easily enough from the rate transition matrix.

5.2.4 Replacement of Zero Reward States

In cases where it is not advantageous to remove states with zero reward and without limiting the model using Beaudry's original technique of making zero reward states absorbing, a proposed workaround is to replace zero rewards with a reward that can be used to closely approximate zero. This replacement reward, ϵ, would be an extremely small number, preferably the smallest number that can be used by the system used to solve the MRM. For instance, ETMCC is written in

the Java programming language, and an ideal number would be `Float.MIN_VALUE`, approximately 2^{-149}[Mic03], the smallest possible number that Java can represent. However using a number as small as this can have an affect on the numerical methods underlying the solution of MRM and this will be investigated further in Section 5.2.5.

Formally expressed, if our original MRM was $\mathcal{M} = \{S, \mathbf{R}, L, \rho\}$ then the modified MRM is defined as $\mathcal{M}' = \{S, \mathbf{R}, L, \rho'\}$ where

$$\rho(s)' = \begin{cases} \rho(s) & \text{if } \rho(s) > 0 \\ \epsilon & \text{otherwise} \end{cases}$$

No matter what size we set ϵ to, it's introduction to the model has the disadvantage of now adding error to the model. If we can estimate this error then we can obtain an idea of the confidence levels on our queries on the modified model.

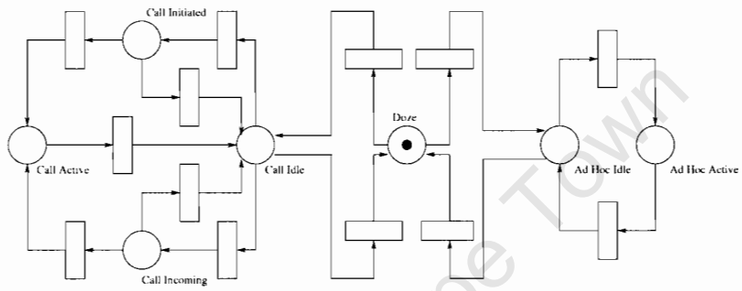

Figure 23: SRN of a mobile station in an ad-hoc mobile network

Example and Observation of Error

To observe the affects of the introduced error, we use a modified model of a wireless mobile station[HCH+02]. The Stochastic Petri Net representation of this model is shown in Figure 23 while the CTMC derived from it can be seen in Figure 24

We have modified the model slightly and introduced a zero reward state by replacing the reward of the initial *doze* state with 0. We then used Monte Carlo simulation to simulate the model over a period of time after the model had been modified to take into account the error. Figure 25 shows the growth of the reward earned for 1000 simulations. Each simulation was terminated after the time limit of 1200 was reached. In this case ϵ was set to 0.00001. The error grows within a set bound as can be seen in the lower graph in Figure 26 and as the time increases the range in which the error falls increases.

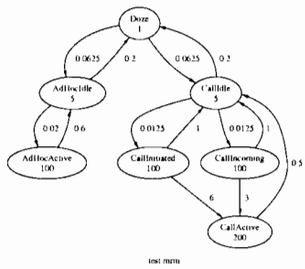

Figure 24: CTMC extracted from Figure 23

Due to the small size of the error in relation to the average reward earned per state we propose that we can determine a simpler estimation of the error by increasing the error earned over time. Instead of assigning ϵ to only states with zero reward we assign it to all states. The justification behind this is that by doing this the error earned each time unit is now constant and therefore linear with respect to time. Our reward structure is now

$$\rho'(s) = \rho(s) + \epsilon$$

with total error accumulated $\epsilon \times t$. In Figure 26 the top graph shows this linear growth with respect to time.

While this method allows for the total error accumulated to be calculated trivially, the error added to the model will grow at a faster rate and therefore affects the length of time before the error becomes significant. This extra error is in addition to the error introduced by ϵ affecting the outgoing transitions of the zero reward states.

However, it must be noted that even the linear growth of error in our second approach still produces a total error that is orders of magnitude smaller than the total accumulated reward of the model. By choosing an suitably small error the error introduced into the calculations could be small enough to not affect the model significantly.

In examples shown in Figures 25 and 26 the zero state reward is only one state in a relatively small model. In models with extremely large state spaces but with a relatively small number of zero reward states it would be more advantageous to only introduce error into the zero rate rewards. When the models residence time in the zero reward state is large, then it is more advantageous to calculate the error using error introduced to all states.

To show the growth of the error in a model with differing number of zero reward states, and therefore

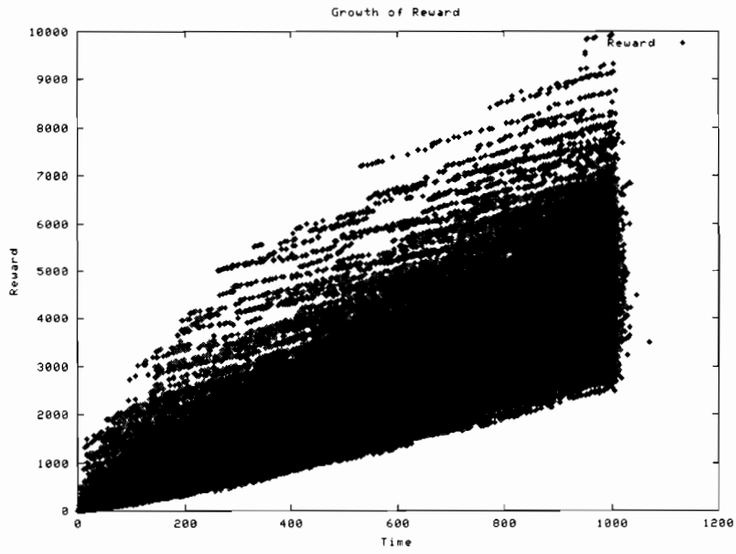

Figure 25: Growth of Reward in CTMC

differing residence times in zero reward states, we simulate a simple grid like Markov chain, comprising a 10×10 grid with states adjacent states linked by double links (both leaving and entering) with equal transition rate . States were either *functioning* or in *failure*. When in *failure* they earned 0 reward per time unit, while being in a *functioning* state earned a reward of 100 per time unit. Initially the ratio of *failure* states to failure states was set to 1:4. The growth of error can be seen in Figure 27 in the leftmost graph. The graph when the ratio was increased to 1:1 (center) and 4:1 (rightmost) are also shown.

As the number of zero reward states increase the distribution of the total error produced by adding error to these zero rate states, tends to increase and finally be limited by the line produced by adding error to all states. This seems to indicate that in certain cases when the total residence time in zero reward states is small relative to other states, the error in the model is best estimated by looking at the actual error growth. In cases where the residence time in zero reward states is large it is simpler to merely use the total error introduced by including error in all states. The error introduced in this manner is in fact the *maximum possible error* of the model.

5.2.5 Additional Numerical Complexity

The ϵ-reward technique has, however, an affect on the underlying numerical methods used in model checking. For our research we use the Erlangen-Twente Markov Chain Checker (ETMCC)[HKMKS99],

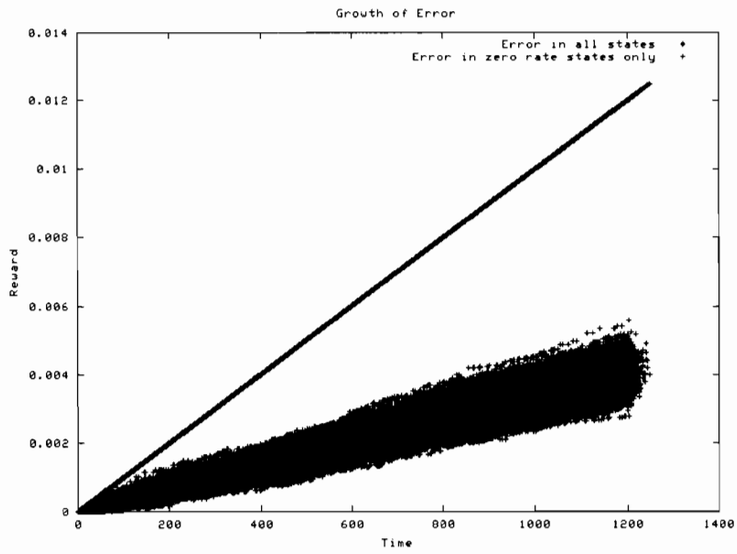

Figure 26: Growth of Error in CTMC

a symbolic model checker, which uses the Fox-Glynn[FG88] method to compute the Poisson probabilities of the underlying Markov chain. The affect on the algorithm is such that the greater the orders of magnitude between the largest and smallest elements in the rate matrix the longer the algorithm must run, and more importantly, the more memory is needed. Figure 28 shows the number of iterations needed by the tool in order to solve sample CRL queries performed on the model described in Section 5.2.4. For small ϵ the number of iterations needed begins to increase quite significantly once ϵ is smaller than 0.0001.

In our example, when attempting to make ϵ smaller by another order of magnitude, we run into memory space problems. Note however, that the execution time remained reasonable, despite the increasing number of iterations, and that memory was the primary constraint. We performed the verification on a 1.4Ghz AMD Athlon processor with 256MB RAM.

In this case where the rate matrix has entries which differ by large orders of magnitude, Runge-Kutta methods might possible be used for better performance[BHHK03].

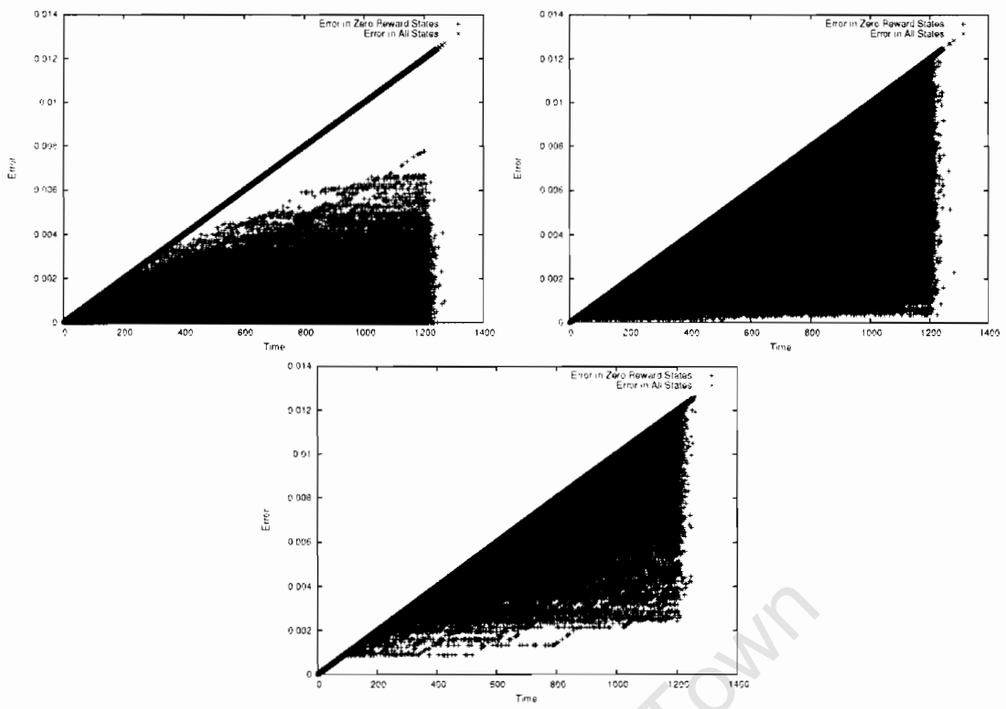

Figure 27: Growth of Error as Number of Zero Reward States Increase

5.3 Limitations of the Logic

5.3.1 Nested Reward Structures

In a CRL expression with nested sub-expressions, the solution to the sub-expressions are not affected by the expression in which they are nested nor by other sub-expressions that are not nested within the sub-expression we are evaluating. In the following expression

$$\mathcal{P}_{\trianglelefteq p_1}(\overbrace{\mathcal{P}_{\trianglelefteq p_2}(a \bigcup_{I} b)}^{\Phi} \bigcup_{J} \overbrace{\mathcal{P}_{\trianglelefteq p_3}(c \bigcup_{K} d)}^{\Psi}) \tag{32}$$

we have two sub-expressions, Φ and Ψ. The solution of either of these sub-expressions is independent of each other as neither expression depends on the other. Each sub-expression, as well as the parent expression, makes use of a reward interval (expressed as I, J and K) which is used to place a limit on the accumulated reward of the model.

The solution to each subexpression is a set of states that satisfy it, and these sets are then used to

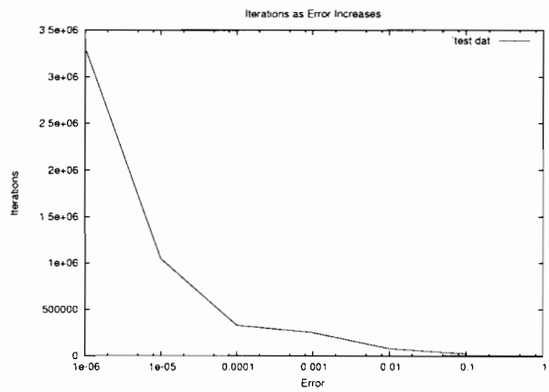

Figure 28: The number of iterations required to solve a Markov chain with ϵ.

solve the larger expression. The solution of each sub-expression would then be independent of the other sub-expressions as well as of the parent expression. From this independence of expressions we could reason that the reward structure for each sub-expression need not be the same as any of the others and as such the usage of multiple reward structures in a CRL expression would be possible.

The usage of multiple rewards would greatly affect the expressiveness of CRL, allowing requirements to be expressed with more then one performance measure. While in CSL it makes logical sense that there is only a singular reward, that being the passing of time, in CRL there is no reason why a requirement should be limited to a single reward structure.

Examples of the usage of queries using nested reward structures are quite varied. In the case of the mobile device described in Figure 23. In that figure the model is given only a single reward structure for power consumption but there might be other reward structures defined for total power used to transmit, total packets sent, total packets received and so on. We might wish to check whether the device will have transmitted a certain number of packets while using a certain amount of power before failure. We would express this as

$$\mathcal{P}_{<0.9}(\mathcal{P}_{>0.01}(active \bigcup_{power:\leq x} idle) \wedge \mathcal{P}_{>0.95}(active \bigcup_{packets:\leq y} idle)) \tag{33}$$

5.4 Limitations of ETMCC

Besides the limitations on modelling with CRL and on the logic itself we are also limited by a number of limitations that are present in the various tools used that perform model checking on CRL. The limitations are present for a variety of reasons ranging from lack of suitable and efficient

algorithms (Solving for Time and Reward) to current algorithms not being able to deal with certain cases (Negative Reward Rates).

5.4.1 Interval Restrictions

In CRL we can apply a restriction on both path operators which restricts the amount of reward that can be earned before the expression is no longer valid. In the general CRL expression using the $\bigcup$-operator of the form

$$\mathcal{P}_{\trianglelefteq p}(\Phi \bigcup_{I} \Psi) \tag{34}$$

the expression is valid as long as Ψ is reached at time t with $t \in I$.

At the time of writing ETMCC only support intervals of the form $[0..t]$, so that (34) is limited to being expressed as

$$\mathcal{P}_{\trianglelefteq p}(\Phi \bigcup_{\leq t} \Psi) \tag{35}$$

This limitations is present mainly due to the fact that while algorithms are available[BHHK03], they are computationally expensive[HCH+02].

However we are not solely limited to model checking with a $[0..t]$ interval as it is possible to solve a CRL requirement for an event that is to happen after a certain time. In this case the interval the statement is to be solved against is $[t..\infty)$ and we can exploit the fact that $P(\overline{\phi}) = 1 - P(\phi)$ where $P(\phi)$ is the probability on CRL requirement ϕ using an interval of $[0..t]$ and $\overline{\phi}$ is the same requirement except using an interval of the form $[t..\infty)$.

5.4.2 Negative Reward Rates

In sections 5.2.2 and 5.2.3 we detailed the removal of zero reward states and the measures taken to prevent the loss of logical properties of the model. In general Markov reward models are not limited to non-zero reward[ST88], however when using ETMCC and specifically when using the Duality theorem to transform a MRM to an equivalent CTMC the usage of negative reward rates would no longer be possible.

While the presence of zero rewards causes division by zero when doing the transform, the presence of negative rewards does not prevent the transform, but the results are most undesirable. In Figure 29 we are shown three transitions. The top transition has a state a with reward -1 which moves to b at rate 4. When the duality theorem is applied, shown in the 2nd transition, the transition rate is now -4 which is analogous to a transition from b to a with rate 4, depicted in the 3rd transition.

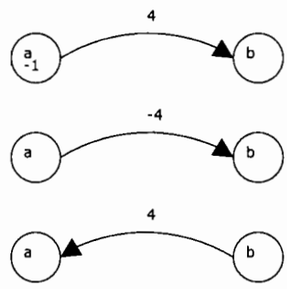

Figure 29: Negative rewards cause transition directions to be reversed

The usage of negative rewards with tool of the nature of ETMCC is also further complicated with the usage of the Fox-Glynn algorithm, which is specifically geared towards computing the probability densities of Poisson processes which are birth only processes. A MRM with a mixture of positive and negative rewards would obviously not fulfill the requirements to be a Poisson process.

When presented with a model with a mixture of positive and negative rewards, a possible approach is to separate the model into two separate models. In the first model we keep the positive rewards as is while reducing the negative rewards to zero, while for the second model we reduce the positive rewards to zero while replacing each negative reward with it's corresponding absolute value. This leaves us with two models, one concerning the accumulation of reward and the other the loss of reward, but both expressed with positive real numbers. The possible large number of zero reward states can be handled using any of the techniques describes in sections 5.2.2, 5.2.4 and 5.2.3.We can now reason independently about each model and as as shown in section 5.3.1 we can combine the two models in larger CRL expressions with out either of the model being affected by the other.

5.4.3 Solving for Time and Reward

So far we have treated the time and reward properties as separate entities, reasoning about them using the separate CSL and CRL logics. In actual fact these two logics are merely components of a single super-logic, CSRL, previously discussed in Section 2.4. The ability to be able to reason about both time and reward would prove to be an invaluable addition.

Although techniques such as simulation can already carry out model checking using CSL (Section 3.2) we do not know of any work that has been done extending this to CSRL. While simulation techniques might provide an easy way to do this, ideally numerical techniques would be needed to overcome the problems inherent in simulation.

Katoen, Hermanns et al. have done work in this area[HCH+02], extending the results obtained in

model checking time bounded $\bigcup$-operators (Section 2.5.5) to time and reward bounded $\bigcup$-operators, which allowed them to show that in order to solve a CSRL equation of the form $\mathcal{P}_{\trianglelefteq p}(\Phi \bigcup_{\leq r}^{\leq t} \Psi)$ on CTMC $\mathcal{M}$ we can solve $\mathcal{P}_{\trianglelefteq p}(\diamond_{\leq r}^{[t,t]} \Psi)$ on the simpler CTMC $\mathcal{M}'$ where $M' = M[\Phi][\neg(\Phi \wedge \Psi)]$ ($M[\Phi]$ is defined in Section 2.5.5).

Investigation was carried out into three potential numerical approaches: using a pseudo-Erlang approximation, discretisation and an analysis of occupation time distributions. However none of these have proven to be truly viable in terms of speed and storage concerns, although research does continue. It should be noted that once a suitable algorithm is developed the additional ability to solve for both time and reward would be a non-trivial addition to performability analysis.

5.5 Conclusion

In this chapter we have identified various limitations on the CSL logic and the tools available. We have focused on the removal of zero rewards from MRM, and shown the problems inherent in wholesale removal of zero states without considering the logical characteristics of that state. We have proposed a method which analyses the requirements on a ETMCC and can successfully remove zero reward states while maintaining logical consistency.

We have also proposed a method where zero reward states are replaced by some small number in order to closely approximate zero. We have shown that the error this introduces can be estimated easily and that it has a set maximum effect on the model. We have also observed the additional numerical complexity places on model checking tools such as ETMCC.

We noted that due to the independence of nested expressions that the usage of multiple reward structures is possible without any further numerical techniques than the ones currently utilised.

Observed limitations on the ETMCC tool are also considered and methods to reduce their impact are proposed. Finally we considered the ability to solve for both time and reward and the efforts to find a suitable numerical method to meet this problem.

Chapter 6

Case Study: Service Management of a Beowulf Cluster

6.1 Introduction

In this chapter we will conduct a case study using CRL and ETMCC to apply various service level requirements onto a working system. Our aim is to ascertain whether MRM and CRL suitable for the tasks of determining whether requirements can be met as well as providing information allowing new requirements to be established.

The system we choose to analyse is a processing node of the ALICE experiment currently being built at CERN. When complete ALICE will generate petabytes of data that must be processed as quickly as possible. Once captured this data is exported to nodes situated at various locations world wide where it is processed. The node situated at the Department of Physics at the University of Cape Town and is comprised of a Beowulf class parallel cluster.

6.1.1 The ALICE Project

A Large Ion Collider Experiment (ALICE) is a collaborative experiment currently being constructed at CERN, with 65 institutions from 29 countries registered as members. The experiment, when completed, will consist of a large detector on CERNs main accelerator ring. The detector's main purpose will be to observe the collisions of Lead nuclei. From these collisions numerous high energy subatomic particles are generated whose properties are then studied.

When a collision occurs, which is known as an *event*, all relevant data related to that collision is captured. The frequency of the events coupled with the size of the data generated would mean

that practical storage solutions would not be feasible. To prevent this the majority of events, which have little chance of containing useful data, are discarded, leaving behind events of interest. Even though most events are filtered out, interesting events are projected to be produced at a rate of about forty events per second. Each event when recorded can will require on average of 25MB of storage, although this can vary up to 2GB.

The experiment, when activated, will run for one million seconds, approximately eleven and one half days, during which time it will generate datasets with a total size reaching in to the petabytes. Once the data is generated it is distributed to various processing and storage facilities belonging to the various members of the ALICE team, where it is made available to researchers to perform analysis.

6.1.2 Beowulf Clusters and Grid Computing

Initially developed by Donald Becker at NASA[SSB+95], the Beowulf cluster is a parallel computer where each node is comprised of an off the shelf desktop personal computer. This is in marked contrast to previous generations of parallel supercomputers which were comprised of specialised components constructed by dedicated manufacturers. Due to the already installed base and availability of off-the-shelf components, Beowulf clusters have become extremely popular at tertiary institutions.

Initially these clusters were simple batch control systems with no redundancy or process control. Jobs were merely submitted into a queue and the results retrieved on completion. To obtain optimal parallelisation and performance applications often had to be rewritten using parallel and messaging libraries. Often applications that did not require any type of inter-node communication merely used shell scripts to distribute data to the various nodes in batch before execution began.

Grid computing is a recent development that has come to the foreground as the number of Beowulf and other commodity based parallel cluster installations has grown worldwide. Grid computing allows a number of separate computing facilities to be linked together and presented to the user as if it were one singular computing facility. When a user submits a job to the grid, the job is distributed to the various computing nodes in a manner that is transparent to the user.

The grid for the Alice experiment is divided into a number of hierarchical *tiers*, with the facility at CERN being regarded as tier 0 where a significant portion of the computing power is located. A user is considered as being at tier 3, while the various processing and storage nodes are either tier 1 or tier 2 depending on their capacity.

6.1.3 Aims and Methods

This case study studies problems in two differing domains, namely a study of the usefulness and ability of MRM/CRL and a performance requirements analysis of a distributed system.

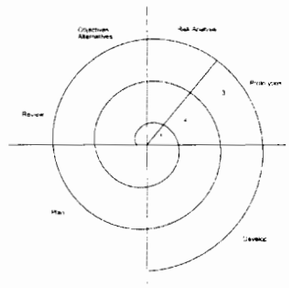

Figure 30: A simple example of Boehm's spiral model

So far no formal requirements have been specified by CERN on any of the computing facilities that are participating in the ALICE project. While the performance of each cluster can be determined using a variety of benchmarks and metrics[BBea94], which take a relatively short time to calculate once the cluster is operational, this is not true for determining the reliability of the cluster as to do so would require the cluster to be run for an inordinate amount of time.

To examine the reliability question we are then left with either simulation of the system or solving the system analytically, which is the approach we use in this thesis.

In modelling and verifying this system we will also attempt to integrate model checking and the usage of CRL into standard software and systems engineering methodologies. In particular we will be following a variation of Boehm's spiral model[Som96]. Figure 30 shows a representation of Boehm's model.

Boehm's model was conceived as a model for the software development process, but we adapt it for usage in modelling. The spiral model fulfils our need in that we start with an initial model and then in each iteration we add to the model until we arrive at our final model.

In our case, we begin with a model of the cluster in it's current configuration, comprising 20 processing nodes. We then apply requirements expressed using CRL on the system. The requirements deal specifically with performance ("*How many events can we process?*") and budgetary ("*How much will it cost once failure occurs*") issues. From this initial model we will then increase the size of the model to represent the proposed next step in the cluster, a 250 node tier 1 cluster. We will then apply the constraints to this model and observe whether the same requirements will hold for this model as they did for the original. The model is then extended again to represent the proposed final configuration of 1000 computing nodes.

6.2 Cluster Specification

6.2.1 Description

Currently the cluster situated in the Dept of Physics only consists of 20 processing nodes and it is on this current configuration that we set requirements and check these against the model. The cluster is projected to grow in the medium term to about 250 nodes which will enable it to be placed in the first tier of the ALICE grid. However this 250 node cluster would not be dedicated solely to CERN but would form part of a larger South Africa wide grid infrastructure open to other tertiary institutions in Southern Africa. In the long term the cluster might grow up to 1000 nodes and become a dedicated supercomputing facility in the university.

In the current configuration each node is directly connected via a switch. This leads to a single point of failure where if the switch fails the entire cluster performs no work. Once the system is enlarged to the envisaged 250 node cluster multiple switches will be part of the system. Another single point in the system is the *master node* of the cluster. This is the node to which jobs are submitted to the cluster and which then distributes the jobs to the nodes in the cluster.

6.2.2 Hardware Specification

The hardware specification of each node in the system is as follows

Component	Make and Specification
Hard Disk Drive	Maxtor/Western Digital 80GB
Motherboard	Asus P4BGL-VM
Central Processing Unit	Intel Pentium IV 1.4 GHz
Network Car	D-Link DFE530TX

The master node of the cluster is the exception, as it has dual processors as well as seven 100GB disk drives which it controls as part of a RAID-5 array. Connecting the master to the other nodes in the cluster is a Surecom EP826D-2T switch.

6.3 Modelling the System

6.3.1 System Characteristics

From the hardware specification in 6.2.2, we can identify which components of the node contribute to the behaviour of the system. The only components that fail due to operational usage are usually

the hard disk drive and power usage. We differentiate here between operational and manufactured failure, where operational failure is failure occurring due to normal usage of the component and manufactured failure is defined as faults that are present in the component from the time of it's manufacture. No other other components has the number of constantly moving parts that are found in these two components, which is why the occurrence of operational failure of a processor or network card is relatively rare.

We can safely discard the affects of operational failure of all the components besides the hard disk drive and the power supply. However we do include failures due to power outages, which occur with enough frequency to potentially affect the performance of the cluster.

6.3.2 Choosing a Modelling Formalism

For this case study we are modelling a system that will vary in size, depending on what iteration of the model we are examining. Of the specification techniques described in Chapter 4, production rule systems (PRS) most closely match our requirements for a formalism. Using PRS will allow us to easily increase the state space size of the model with little user intervention, while still keeping the structure and properties of the model consistent.

The main characteristic of this particular model which lends itself to the usage of PRS is that of the general uniformity of the model. Whether we are modelling the initial configuration or the final larger one, the underlying structure of the two models will remain functionally equivalent. By changing a few numbers in the specification of the initial MRM we can then generate the larger iterations.

We use the PRS we implemented, which we described previously in Section 4.3.3.

6.3.3 Constructing the Model

The underlying structure of the model is quite simple and linear in nature as shown in Figure 31. Each state is connected to it's neighbour state by two transitions which represent failure and repair.

The failure transition rate is dependent on the number of processing clusters and their failure rates. The hard disk drives used in our cluster have a *mean time before failure*(MTBF) of 50 0000 hours. The power supplies in each node varied in MTBF between 50 0000 and 100 0000 hours and we took the average of 75 000 hours for each units MTBF.

The cluster at the moment has no uninterruptible power supply and so is vulnerable to power outages. Already this year the campus at UCT has suffered from four total power failures. When these occur the entire cluster is rendered inoperable. To represent this each state has a transition leading to the *failure* state. We estimate that on average there are 3 total power outages per year.

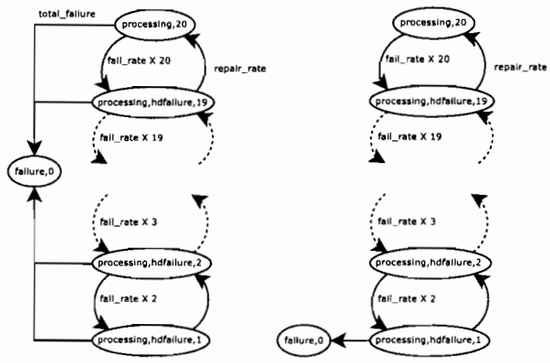

Figure 31: Markov model of cluster with total power failure included on the left and without on the right

6.3.4 Reward Structure

The reward structure of the cluster is dependent on the number of functioning nodes. We have estimated that on average it takes each node 4 hours to process an event. A 20 node cluster would be able to process on average 5 events per hour.

When determining the processing capacity of parallel systems we are often subject to Amdahl's Law[Bro00] which describes the speedup we can expect by parallelising a process. It is described as

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p/N} \tag{36}$$

where $T(N)$ is the time required to complete a task using N processes, T_s is the serial portion of the task and T_p is the parallelisable portion. In our case Amdahl's ratio is not constant due to the fact that no two jobs processed will have the same ratio.

To simplify we will assume that jobs submitted to the cluster are perfectly parallelisable and that the processing ability of the cluster scales linearly as it grows. This will give us a best case scenario with which to model the cluster. We can then guarantee that if a requirement is not satisfied by this model, it will not be satisfied by any others.

6.4 Constructing Requirements

6.4.1 Determining Requirements and Service Levels

We briefly mentioned in Section 6.1.3 that the ALICE consortium does not specify any requirements on the reliability or performance. Any member of the ALICE consortium should not undertake to

become a member of the grid without the necessary facilities and budget. However without set requirements for the cluster, potential members of the grid can only estimate the size of the cluster needed and what type of components are needed for each node. This can potentially lead to over spending in the construction of a cluster which meets the same requirements as that of a potentially cheaper cluster.

The setting out of requirements from the outset would also allow the consortium to determine what tier status each cluster should receive. By setting out in a rigorous fashion the requirements needed to be assigned to specific tier groups would allow the management of the grid to enforce a *service level agreement*(SLA) between themselves and the various members of the grid. Using the individually assigned service levels, the grid can then dynamically assign processes and jobs[CFK+02] according to the requirements of each process and the services available at that particular time.

In fact it would seem that MRM coupled with a performability logic such as CRL offers an ideal tool for the specification, modelling and verification of SLAs. CRL offers a formal language to rigorously define the expected behaviour of the system that can be applied right throughout the capacity planning process. To the service provider (the individual cluster) it allows for the system to be modelled using MRMs and then checked before implementation. Once requirements that they believe can be met on the system are specified they can negotiate with the client (in our case the grid management) whether these requirements are sufficient. Once a SLA has been accepted by both parties, the management can monitor the performance of the cluster and check whether the performance still meets the requirements set out in the SLA.

The drawing up of SLAs is part of a larger process of *capacity planning*, which is required in order to determines the systems *adequate capacity*. Menasce and Almeida[MA98] specify that the three main elements needed to define adequate capacity are:

- Service level agreements
- Specified technology and standards
- Cost constraints

We have discussed how CRL and MRM allow us to specify and verify SLAs but in fact the versatility of both CRL and MRM allow us to also get an idea of the cost constraints of the proposed cluster. By substituting a reward structure to model the cost of failure (new components, repairs) we can check whether the proposed cluster will be too costly to operate. This is quite relevant for Beowulf clusters, which are constructed out of cheaply available commodity hardware as discussed in 6.1.2.

There is a trade off between the cost of the cluster initially to construct, and the maintenance required once the cluster is operational. With custom built parallel processing systems the initial cost was

Probability	No. of Events Processed
99.9	10 000
98	100 000
95	500 000

Table 1: Performance requirements placed on the cluster

high but due to the specialised, custom built components used. The opposite is usually found in Beowulf clusters where due to mass production purchase costs are kept low, but reliability does not reach the levels of custom built specialised hardware. The costs associated with running a Beowulf cluster are therefore not constant, and depend on the failure rates of the hardware components and the rate at which they are repaired. Again, CRL lends itself to this task allowing the cluster operator to obtain confidence levels as to the budgetary requirements of the cluster once it is operational.

Due to the lack of requirements specified by the ALICE management, requirements were determined through discussion with the cluster team at UCT. The requirements determined by them are what they believe are needed to allow the cluster to run under budget and to be reliable and powerful enough to be part of the first tier of computing facilities in the ALICE project.

6.4.2 Translation to Continuous Reward Logic

Performance of the cluster is measured in how many events are processed by the cluster. Due to CRL being a performability logic we will be looking at the the interplay between the reliability of the cluster and how this affects it's performance. The first metric interest is the distribution of reward till failure[BHHK00b]. This is characterised as the probability of going from a functioning state where reward is earned, to a state in failure where the reward earned is zero. For the model of the cluster it can be expressed as

$$\mathcal{P}_{>p}(processing \bigcup_{>I} failure) \tag{37}$$

where *processing* is the atomic proposition assigned to states where reward is earned and $failure$ is assigned to states where no reward is earned. p is the probability that the requirement will be met and I is the number of events processed.

In drawing up the requirements it was decided that the cluster should be able to offer 3 levels of confidence, with each level corresponding to a certain number of events that will be processed before some failure occurs. The values of the confidence level (p) and the processed events (I) are given in Table 1.

The initial value of I is set to 10 000 as this is the minimal number of events needed to ensure that there is at least a 99% confidence level that the the conclusions reached from the processed events are in fact correct. The selection of 10 000 events as the minimum needed is derived from the standard deviation of the Poisson distribution, which governs the detection of events. If v events are detected the standard deviation is simply $\sqrt{v}$. The *fractional uncertainty*[Tay82] is then defined as

$$\frac{\sqrt{v}}{v} = \frac{1}{\sqrt{v}} \tag{38}$$

For us to get a fractional uncertainty of less than 1%

$$\frac{1}{100} \geq \frac{1}{\sqrt{v}} \tag{39}$$

and therefore

$$v \geq 10000 \tag{40}$$

This then represents the absolute minimum number of events that must be processed. Any failure before 10 000 events being processed represents an undesirable performance level. The two latter numbers represent performance levels that are more ideal and which should be sufficient for the majority of jobs run on the cluster.

We also set requirements on the budgetary constraints of the cluster. For this requirement we use a modified version of the model, where instead of modelling outages caused by power failure we are more concerned with the total cost of running the cluster over a period of time. We replace the $failure$ state with a $finished$ state. This state, like the $failure$ state, is reachable from every other state, with the transition rate set so that the mean time before the transition is taken is approximately one year.

These cost requirements are expressed in CRL as

$$\mathcal{P}_{>p}(processing \bigcup_{<I} finished) \tag{41}$$

For this requirement we will use a new reward structure which representing the costs incurred for every hour that we are in a state where there is one or more hard drive failures. This cost is incurred by the combination of hourly repair costs, the total which is dependent on the time spent in the state, as well as the cost of new equipment. The interval I representing the total cost to repair the cluster over a year and the corresponding confidence levels are shown in Table 2 for the initial 20 node cluster.

The values in Table 2 are determined by budgeting approximately R1000 for every node in the cluster per year. This covers the replacement of all equipment in the cluster as well as costs associated with the labour required to do so. We use three levels of confidence (as in Table 1) and with each corresponding confidence level associate some monetary amount which we hope to not exceed during that year in operation. We set a reasonable confidence level that we will meet our budget of 95%. However budget overruns do occur and we take this into account by setting two more confidence levels, one for a 25% overrun which we set a level of 98% for and a 50% budget overrun with confidence level 99.9%.

Probability	Repair Cost
99.9	30 000
98	25 000
95	20 000

Table 2: Budgetary requirements on a 20 node cluster over the course of one year

The instantaneous cost gained by the purchase of of new equipment is usually characterised as an *impulse* reward which is earned when the transition associated with that reward is taken. Solution techniques to MRM with impulse rewards are available[QS94, RT99] however ETMCC lacks these techniques.

We can overcome this lack of solution techniques in our tool through the usage of specific modelling techniques, which we detail in Section 6.5.3.

6.5 Results and Discussion

6.5.1 Expressiveness of CRL and MRM

As part of the case study we examine the usage of MRM and CRL and how they affect the construction of the model. We will focus on the expressiveness of the model and the requirements, and what ramifications any limitations on either will have on the model.

MRM

When using MRM it is often not advantageous to refer directly to states by their unique identifiers, which is why it is almost imperative to be able to use state labellings. State labellings place a layer of abstraction on the model making it easier to reason about the model. This became apparent when we consulted with the Physicists who run the cluster, and during our conversations they could

easily create their own state labellings to reason with, following which the model could be created and the state labellings applied.

There are limitations on our MRM due to there being limitations on the tool we use to solve the model. As mentioned earlier in Section 6.4.2 we are limited to state rewards and we have to incorporate the impulse rewards into the state reward of the model.

CRL

The abstraction of the model via state labellings also has great use when determining CRL requirements. As with MRM we do not have to deal with individual state identifiers making the process of drawing up requirements simpler, however without sacrificing any expressiveness.

Another issue when using CRL is the applicability of each particular CRL operator to the model. For this case study, we found that all requirements were expressed using the probabilistic CRL operator ($\mathcal{P}_{\trianglelefteq p}$) coupled with the reward bounded until operator ($\bigcup_I$). It seemed to us that the steady state operator ($\mathcal{S}$)is not needed when examining requirements of this nature.

We also found we could reduce the complexity of CRL requirements by replacing state labellings that are connected via the logical state operators ($\vee, \wedge, \neg$) with a new state labelling. This is already what ETMCC does internally, but by expressing it beforehand we can make it extremely clear to those specifying the requirement what states are to be involved.

We also must restate here the current difficulties in solving CSRL, the parent logic of CRL, which we detailed previously in Section 5.4.3.

6.5.2 Performance Requirements

20 Nodes

Substituting the values in Table 1 into CRL equation (37) and applying them to the models of the cluster shown in Figure 31 produced the graph in Figure 32.

This graph shows the final probability of a state meeting the requirement. The numbers on the x-axis represent the state that has x hard drives initially. As is clear from the graph the probability that any state will not satisfy the requirement is extremely high. For all three requirements specified the the lowest probability computed is about 0.5, well above the limits of 0.001, 0.02 and 0.05.

It is quite clear that we will not be able to meet any of the requirements set with this model. Even for the minimum performance level of 10000 events we can at best offer a success rate of 50%. For the 100000 and 500000 event performance levels the probability of meeting the requirements is

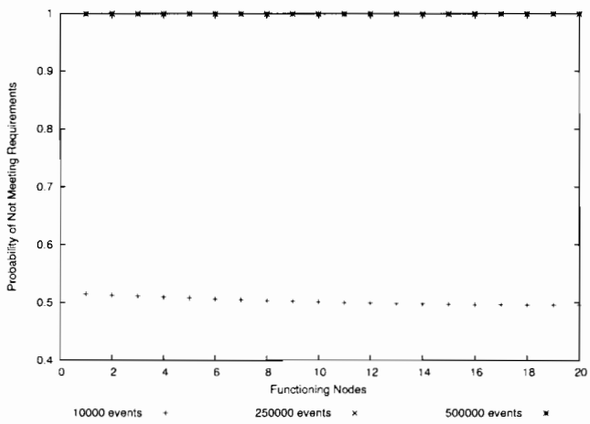

Figure 32: Probabilities of a 20 node cluster, with power failure, fulfilling set performance requirements

extremely close to zero. It is clear that the minimum level of events processed must be lowered in order to be able to guarantee with a high probability that the requirements will be met.

Figure 33 shows the probabilities of being able to process x events before failure, where x is a number of minimum performance levels. Even halving the number of events needed to be processed to 5000 only produces a probability of 0.3 that the requirement will not be met. Further reducing the number of events needed to 2500 produces a probability of between 0.2 and 0.15 that requirements will not be met.

Applying the same requirements but to a model of the cluster without power failure produces Figure 34. This plot shows the marked difference the lack of power failures has on the cluster. Instead of failing the requirements as in Figure 32 the requirements are easily met.

This initial result indicates that in clusters with relatively small number of nodes, in order to meet the performance levels required uptime is crucial. Even the sparse power outages we placed in the model, produce results indicating a near 100% failure rate before requirements are met. To overcome this the cluster will require some manner of backup power supply to ensure the required number of events are processed before failure.

The affects of a power failure on a cluster of this size are quite significant, and illustrate the fact that external factors need to be taken into account. As we saw when we excluded power failure from the model. The difference in Figures 32 and 34 shows that a factor that is often taken for granted, uninterrupted power supply, can have drastic affects when taken into account.

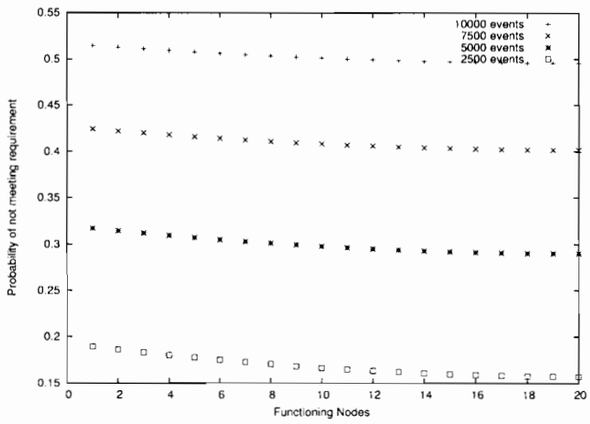

Figure 33: Probabilities of meeting requirements with lowered minimum performance levels

250 Nodes

We extend the model to represent the proposed second phase of the cluster, which is composed of 250 nodes. We again model the cluster using two scenarios to take into affect the presence of total power failures as well as the lack there of. With the larger number of nodes we expect performance to be markedly better but what effects a total power failure will have on this model remains to be seen.

Figure 35 shows the probabilities of meeting the various requirements when power failure is factored in. Of the three requirements, only the plot of probabilities for the requirement of processing 10000 events seems to potentially meet the requirements. Even when starting with only 1 functioning hard drive, there is still a close to 70% probability that the cluster will be able to meet the requirements, while when it is fully functional this increases to more than 90%. This allows us to set a broad confidence intervals on the cluster processing the required 10 000 events, although the odds of the cluster being reduced to a single hard drive are extremely small.

However ideally we would like cluster to have a 99.0% chance of meeting it's requirements (p set to 0.001) and in order to do this we need to adjust I to a level where the requirements are met. Figure 36 shows the probability of meeting the requirements with I being lowered to the same levels we used in Figure 33. With the increased computational power of this largest cluster it seems that the probability of not being able to process 10000 events is just above 0.05. When the cluster is fully functioning reducing the minimum number of events, as we did for the 20 node cluster, does not have that of a significant as it did with the smaller configuration. However the differences in the probabilities for each performance level increases as the number of non-functioning nodes increases.

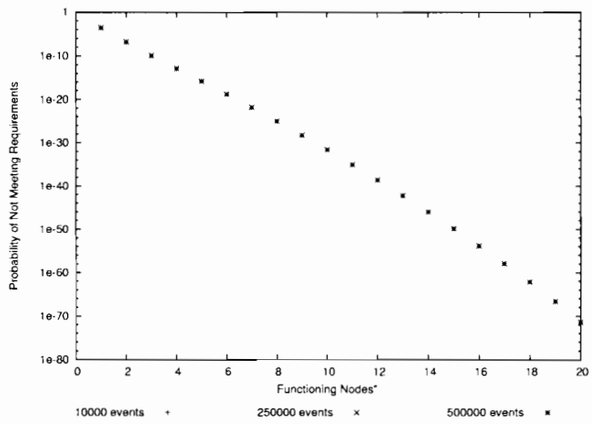

Figure 34: Probabilities of a 20 node cluster, without power failure, fulfilling set performance requirements

If the cluster is forced to operate with a large number of non-functioning nodes, it might be better suited to offer a service level based on one of the lowered minimum service levels. However because the cluster rarely starts processing with a significant portion of non-functional nodes we can focus on the probabilities of most functional states, those states which have 240 - 250 nodes functional.

Figure 37 shows the probability of each state satisfying the set requirements, using a model that has no power failure. From the graph it is clear that our requirements are easily met. Even with only one operational hard drive we meet the requirements and for cases where we have more than 20 nodes operational the probability is so close to zero that underflow occurs and the probability that the requirements will not be met becomes zero.

Again in this larger configuration the affect of the possibility of total power failure is still present. For the minimum service level of 10000 events we approach our desired goal of 99.9% percent, however for the 100000 and 250000 event level we can only give a confidence level of slightly below 50% and 10% respectively. The larger number of nodes has had little affect on the ideal service level of 250000 events processed, and this service level is still governed more by the affect of power failure than the processing power of the cluster.

1000 Nodes

Finally we extend the model to represent the final phase of the cluster, that of the proposed supercomputing facility incorporating 1000 processing nodes.

Figure 38 shows the probability distribution of a 1000 node cluster modelled with power failure.

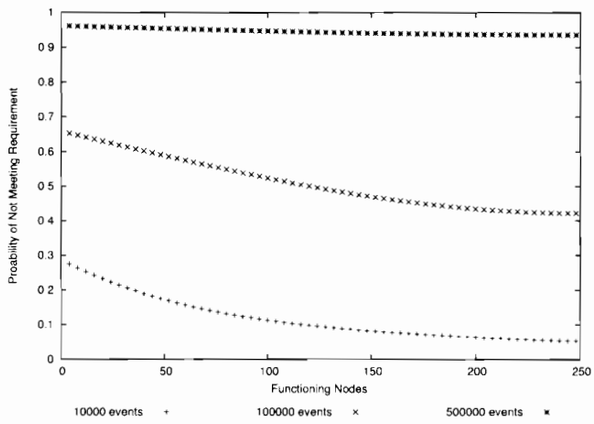

Figure 35: Probabilities of a 250 node cluster, with power failure, fulfilling set performance requirements

For the minimum service level we are quite close to meeting requirements, and in practicality the probability computed for not meeting this performance level, 0.0136, is more than adequate. Figure 39 shows the probabilities when we lower the minimum service level and we observe that lowering the minimum-service level has negligible effect when the cluster is fully functional.

For the mid-level performance level of 100000 events processed before failure we notice a marked drop from a probability failure from 0.5 in the 250 node cluster to just above 0.1 in this configuration which is reasonably close in practical terms to our desired confidence level of 0.02.

However even in this final configuration of 1000 possible processing nodes, the ideal confidence for the 500000 event processing level is still not met by quite a margin, 0.5 versus the desired 0.05. Again we can factor this to the significant affect of total power failure which still has considerable affect on requirements that take a large amount of time to complete, such as is found in this requirement level.

There is no need to show the plot for when power failure is accounted for as from Figure 37 we can see that there is practically a 100% success rate for any cluster comprising more than 20 nodes. This will also hold for a 1000 node cluster and the plots will be identical.

6.5.3 Cost Requirements

As detailed in Section 6.4.2, we also set requirements on the cost of running the cluster, using CRL equation (41) and values in Table 2. We again use the same methods in Section 6.5.2, observing placing the requirements on three different models representing nodes with 20, 250 and 1000 clusters.

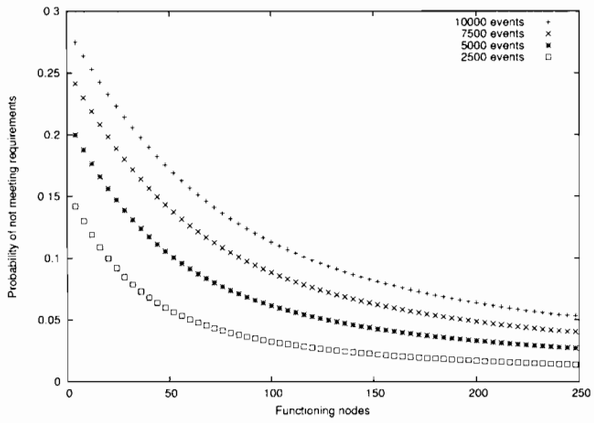

Figure 36: Probabilities of meeting requirements in a 250 node cluster with lowered minimum performance levels

However we will not model the cluster with power failure as this will have no affect on the rate at which hard drives are repaired.

We use the following reasoning to determine the reward structure. The cost to repair a node is composed of the hourly rate cost of labour, r_l, as well as the cost of actual components r_c replaced. Because r_c is an impulse reward and so we need to spread it's cost so that we can solve it using ETMCC.

$$r_h = r_l + \frac{r_c}{t_r} \tag{42}$$

where t_r is the mean time before repair. Setting r_l to 50, r_c to 900 and t_h to 12 hours gives us an r_h of approximately 70. Because the repair rate is constant no matter how many drives are non functional the reward rate is constant as well. Our reward structure for this model is then simply

$$\rho(s) = \begin{cases} 70 & s \models hdfailure \\ 0 & \text{otherwise} \end{cases} \tag{43}$$

While this does not portray the model as accurately as could be done using impulse rewards, for our purposes it is accurate enough to model the cost associated with failures in the cluster.

20 Nodes

For a 20 node cluster our annual budget amounts to roughly R20 000. The budget and confidence levels that we will be within the budget at the end of the year are given previously in Table 2. Using

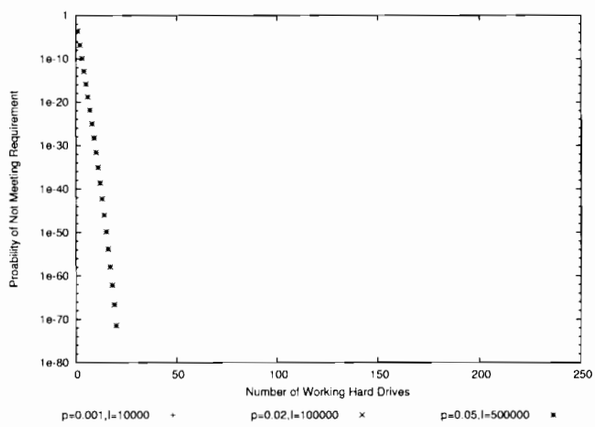

Figure 37: Probabilities of a 250 node cluster, without power failure, fulfilling set performance requirements

these requirements along with CRL requirement (41) produces the three plots in Figure 40

From Figure 40 we note the following. The probability that we will be within our worst case budget, $I = 30000$, is quite high ranging between 0.95 and 0.99. Three states, when we are fully functional or there are either one or two non-functioning nodes, meet the full requirement, that is the probability of being in those states and being within the limit by the time we reach $finished$ is greater than 0.95. Given that the probability of having more than two failed hard drives is quite small and that even for this requirement the we can state with a good degree of confidence that this requirement is met.

For our second requirement where the budget overrun is 25%, $I = 25000$, 10 states meet the full requirement. This indicates that we can be extremely confident that this requirement will be met by the cluster. We could have potentially half of the cluster unavailable at the beginning of the year and it could be repaired and function normally for the rest of the year while still operating within the requirement.

For the ideal requirement, where the annual budget of R20 000 is not exceeded, 8 states meet the total requirement. Again this is indicative that a significant portion of the cluster can fail or be replaced while still operating within the annual budget. However unlike the case where $I = 30000$ where even if the majority of the nodes become non-functioning the probability of meeting the budget still remains greater than 0.95, in this scenario as more nodes become non-functional the probability of meeting the requirement drops quite rapidly, with the minimum probability computed being around 0.61.

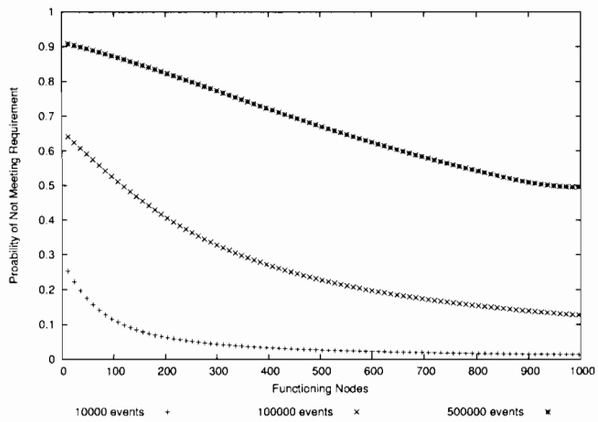

Figure 38: Probabilities of a 1000 node cluster, with power failure, fulfilling set performance requirements

250 Nodes

As the requirements in Section 6.5.3 are met, we scale up the model to represent a 250 model cluster. The budget for this cluster also increases as do the budget overruns. Table 3 lists the budgets and their associated confidence levels.

Probability	Annual Repair
99.9	375 000
98	312 500
95	250 000

Table 3: Budgetary requirements on a 250 node cluster over the course of one year

Substituting the data from Table 3 into CRL requirement (41 produces the three plots in Figure 41.

For the case where $I = 375000$, over 65% of the states in the model satisfy the requirement. Even the states that do not meet the requirement fully still come quite close to doing so with the minimum probability found in any state being 0.975. This is quite a encouraging figure as it indicates that during the year large sections of the cluster can be replaced or repaired while still almost guaranteeing that any budget overrun will be no more than 50%.

The plot for when $I = 312500$ is even more encouraging with 74% of the states in cluster meeting the full requirement. The states that do not meet the requirement only do so by a small amount, with the least successful state still having a probability of 0.925 of meeting the requirement. Again this indicates a high level of confidence that we will not spend more than the specified amount annually.

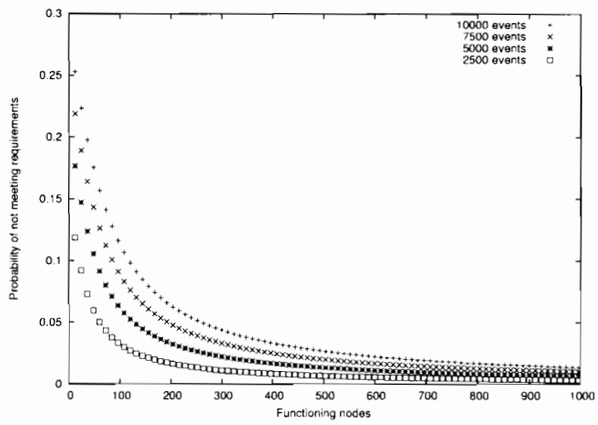

Figure 39: Probabilities of meeting requirements in a 1000 node cluster with lowered minimum performance levels

Finally for the ideal case where $I = 250000$, the number of states that satisfy the requirements comprises only about 60% of the total number of states. This still indicates a high confidence level with a large number of nodes being non-functional while still meeting requirements. However unlike the previous two cases, the probabilities of states that do not meet the requirements begin to drop quite rapidly as the number of non-functioning nodes increases.

1000 Nodes

Extending the model further to the final 1000 node phase, we produce the values for the annual budget and associated confidence levels in Table 4.

Probability	Annual Repair Cost
99.9	1 500 000
98	1 250 000
95	1 000 000

Table 4: Budgetary requirements on a 1000 node cluster over the course of one year

Using these values in CRL requirement (41) produces the plots found in Figure 42.

In this case, the plots for $I = 15000000$ and $I = 1250000$ are extremely close together. For $I = 1500000$, 478 states satisfy the full requirement while for $I = 1250000$ 804 states satisfy that particular requirement. These both indicate that the requirements will be met on this model. Indeed, for both plots the minimum probability of a state meeting the requirements is 0.96 and 0.958

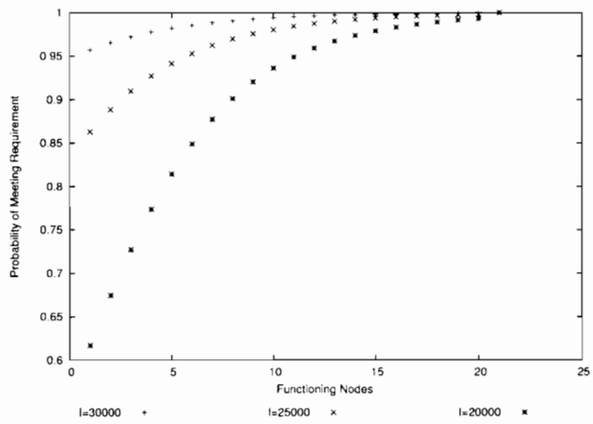

Figure 40: Budgetary requirements on a 20 node cluster over the course of one year

respectively.

For the case where $I = 1000000$, 659 states fully satisfy the requirements. We again notice that after a certain number of nodes are non-functioning the probability of the state satisfying the requirements begins to decrease more rapidly. However unlike the previous two cases which have a minimum probability of meeting the requirements in the 0.60 - 0.65 range, in this case the minimum probability is 0.82, which overall gives the model a favourable confidence that we will be within budget even if a large majority of the nodes in the cluster become non-functional.

6.5.4 Discussion

Performance

It is quite clear from the results obtained that the presence of power outages has a significant affect on the performance on all phases of the cluster. When power failure is taken into consideration both the 20 and 250 node cluster fail all their requirements by a large margin. Only the final 1000 node cluster comes relatively close to meeting only it's minimal and mid-level requirement.

When power failure is taken out of the equation all models easily meet their requirements. This indicates that by far the largest cause of failures are external to the cluster, and as such preventative measures, especially in the 20 and 250 node phases, are needed.

Without some way of providing uninterruptible power to the cluster, the service levels that can be guaranteed by the cluster will have to be lowered significantly from their original estimation.

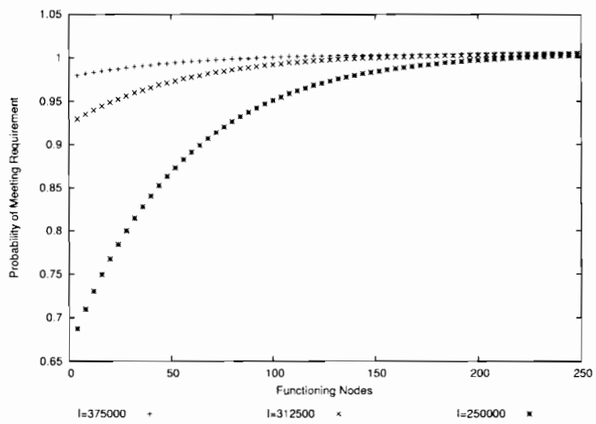

Figure 41: Probability of a 250 node cluster performing within budgetary requirements

Cost

In terms of cost, using current requirements fall within the annual budget of R1000 per node. It is also clear from that the increase in budget need not be linear to the increase in cluster size.

The results obtained from the requirements also enable us to determine how much upgrading of equipment per year can take place while still staying within our budget. In all phases of the cluster we can replace over half our nodes and still state with 90% certainty that we will be in budget.

6.6 Conclusion

We have found the tools and formalisms used to be adequate in allowing us to model the system and verify the requirements. The limitations on CRL, MRM and ETMCC did not provide difficulty. The abstractions provided by state labellings allowed non-specialists to be able to reason successfully about the model and apply constraints to it.

The usage of PRS was also found to be adequate as they allowed for quick and easy generation of a number of models varying in size.

The ETMCC tool provided enough data to enable us to effectively gauge the projected performance and cost of running the cluster, enabling us to reason about whether performance and budgetary goals can be met. By varying the model structure we were able to determine which factors contribute the most to the degradation in performance of the model.

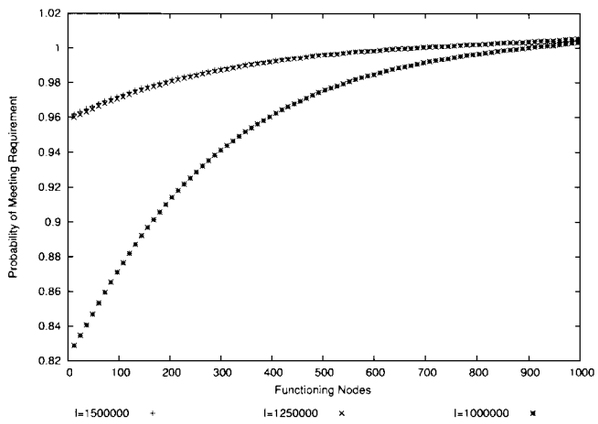

Figure 42: Probability of a 1000 node cluster performing within budgetary requirements

The limitations discussed in Chapter 5 only marginally affected the implementation of this case study, with the few zero reward states generated being easily removed. For other models of systems of this type where there is sustained periods of uptime and reward accumulation these limitations do not seem to be that much of a hindrance. In system models where there is a sustained period of residence in zero reward states and where there might be problems removing these zero reward states, these limitations will have more impact and will need to be addressed using the algorithms described.

Chapter 7

Conclusion

7.1 Summary

In this thesis we have investigated the role of Markov reward models and model checking in the context of service level management.

We investigated the various specification techniques available. Due to the close coupling between CSL and CRL, due to the Duality theorem, the advantages gained and formalisms available through modelling with vanilla CTMCs can be easily transferred to MRMs. We also implemented a production rule system using a high level language interpreter as a proof of concept and showed the advantages in implementation time and code size and complexity in doing so.

The limitations still present in CRM and MRM were also discussed. We placed emphasis on the removal of zero reward state from MRM and proposed two solutions to overcome this. We also found that the ability to model check with multiple reward structures under certain conditions seems to be allowable.

In our case study, performed in Chapter 6, we implemented a cost and performance model of a distributed computing cluster. Our intention here was to successfully use the tools we developed during the course of this thesis, namely the extensions to ETMCC and our production rule MRM generator, to successfully implement a service level model of the system. We also ascertained that the CRL logic was expressive and powerful enough to describe the requirements placed on the system.

7.2 Future Work

7.2.1 Model Checking of both Time and Reward

Section 5.4.3 dealt with the current limitations preventing the verification of both reward and time constraints simultaneously on a MRM. In that section we detailed the various algorithms that have been investigated so far with varying degrees of success.

Algorithms based on the solution of fluid-stochastic Petri Nets[HKNT98] have been proposed as a possible verification procedure for CSRL requirements and work on this continues, mainly at the FMT group at the Department of Informatics, University of Twente.

If suitable techniques are found, it would significantly extend the framework for verifying performability properties.

7.2.2 CSL as an End-to-End Service Level Agreement Specification Language

Currently the usage of performability logics has been limited to the time before implementation with their usage being limited to verifying models of the systems. CSL and CRL provide a convenient, formal specification which has the potential to be used as a language to specify SLAs. However in order to be able to do this these requirements need to be applied not only to models of the system before implementation but to the performance of the system once it is in production.

In Section 3.2 we discussed the ProVer tool, a CSL model checker which uses acceptance sampling and simulation. If instead of using simulation to drive the acceptance sampling tests we used events and data generated from measurements gained from the system we would be able to verify this data via CSL using the algorithms devised by Younes and Simmons[YS02]. This would enable CSL to be implemented as a specification for service level agreements through all phases of the service management process from initial design to service level monitoring once the system is in production. If we refer back to Figure 2 in Chapter 1, this will allow CSL to be applied not only in the off-line planning phase but later in the SLA monitoring and management phase.

The first apparent challenge is to determine the number of samples that are needed to verify the probability within specified error bounds. The number of samples measured from the system might not be sufficient to place the confidence levels within the set requirements. Younes, Simmons and Musliner have tackled this problem, albeit this result is derived from a wholly separate problem domain of AI planing, and have developed an algorithm allowing anytime verification[YMS03] which allows a confidence level to be determined at anytime during the verification process. While this

does not guarantee the confidence level is within the required bounds it does give us an indication of how confident we can be of the result.

The above algorithm is also applicable at the moment to formula of the form $\mathcal{P}_{\geq p}(\rho)$ where ρ is a simple sub formula containing no further $\mathcal{P}$-operators. While having nested formula is not common in CSL, this limitation will have to be addressed further.

7.2.3 Modern Specification and Modelling Formalisms

In the past five years there has been a marked rise in the popularity of the Unified Modelling Language (UML)[BRJ99]. While primarily focusing on the specification and correctness aspects of modelling a system, with the emergence of Real-Time UML[Gro02] performance and dependability analysis is now possible at least in the time domain using CSL.

Over the past two years with there has been significant work extending UML to handle probabilistic[JHA02] properties as well as mapping UML to formal performance[LQV01] and correctness[BDM02] languages and tools and formalising it's structure[MC01]. A remaining problem that will need to be addressed is a suitable extension to UML to enable the modeling of rewards in order for performability analysis to be carried out.

This opens up an interesting avenue of interest where the formal high level specification of UML can be combined with the requirements specification of CRL allowing for UML models, or at least CTMC extracted from UML specifications, to be model checked using the tools and techniques outlined in this thesis. With the main emphasis on UML being a software engineering tool, the work being done here could be used in the application of performability analysis techniques to the software engineering domain.

Bibliography

[AH99] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 14(1):7–48, 1999.

[ASSB96] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time markov chains. In *Computer Aided Verification*, volume 1102 of *LNCS*, pages 269–276, 1996.

[Bag] Doug Bagley. The great computer language shootout.
Online: `http://www.bagley.org/~doug/shootout/craps.shtml`.

[Bai99] Christel Baier. On algorithmic verification methods for probabilistic systems. Habilitation Thesis, University of Mannheim, 1999.

[BBea94] E. Barszcz, D. Bailey, and J. Barton et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, 1994.

[BCC+92] A. Blakemore, P. F. Chimento, G. Chiola, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. *Linear Algebra, Markov Chains and Queueing Models. IMA Volumes in Mathematics and its Applications*, 1992.

[BDM02] S. Bernardi, S. Donatelli, and J. Merseuer. From uml sequence digrams and statecharts to analysable petri net models. In *Proceedings of the third international workshop on Software and performance*, pages 35–45, Rome Italy, 2002. ACM Press.

[BdSM87] S. Berson, E. de Souza, and R.R. Muntz. An object oriented methodology for the specification of markov models. Technical Report CSD-870030, UCLA, 1987.

[Bea78] Martin Beaudry. Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, 27:540–547, June 1978.

[BHHK00a] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model checking continuous-time markov chains by transient analysis. *Annual Symposium on Computer Aided Verification*, 2000.

[BHHK00b] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. On the logical characterisation of performability properties. *Automata, Languages and Programming*, pages 780–792, 2000.

[BHHK03] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29(6), June 2003.

[BK95] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory.* Vieweg Verlag, 1995.

[BK98] Christel Baier and Marta Z. Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, 66(2):71–79, 1998.

[BKH99] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *International Conference on Concurrency Theory*, pages 146–161, 1999.

[BMR02] E. Bouillet, D. Mitra, and K. Ramakrishnan. The structure and management of service level agreement in networks. *IEEE Journalon Selected Areas in Communications*, 20(4), May 2002.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Users Guide.* Addison Wesley/ACM Press, 1999.

[Bro00] Robert G. Brown. Maximising beowulf performance. In *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:244–263, 1986.

[CFK+02] Karl Czajkowski, Ian Foster, Carl Kesselman, Volker Sander, and Steven Tuecke. SNAP: A protocol for negotiating service level agreements and coordinating resource management in distributed systems. In *Proceedings of the 8th Workshop on Job Scheduling Strategies for Parrallel Processing*, 2002.

[CGN95] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. Distributed state-space generation of discrete-state stochastic models. Technical Report TR-95-75, College of William and Mary, 1995.

[CM96] G. Ciardo and A. S. Miner. SMART: Simulation and markovian analyzer for reliability and timing. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, Urbana-Champaign, IL, USA, September 1996. IEEE Comp. Soc. Press.

[CMST90] Gianfranco Ciardo, Raymond Marie, Bruno Sericola, and Kishor Trivedi. Performability analysis using semi-markov reward processes. *IEEE Transactions on Computers*, 39(10):1251–1264, 1990.

[Con03] International Engineering Consortium. Service level management. Online: `http://www.iec.org/online/tutorials/service_level/`, 2003.

[FF97] James A. Fitzsimmons and Mona J. Fitzsimmons. *Service Management: Operations. Strategy and Information Technology.* Irwin/McGraw-Hill, 1997.

[FG88] Bennett L. Fox and Peter W. Glynn. Computing poisson probabilities. *Communications of the ACM*, 31(4):440–445, 1988.

[Gro02] Object Management Group. UML profile for schedulability, performance and time specification. Online: `http://www.omg.org/uml/`, 2002.

[HCH+02] Boudewijn Havekort, Lucia Cloth, Holger Hermanns, Joost-Pieter Katoen, and Christel Baier. Model checking performability properties. *Proceedings of the International IEEE Conference on Dependable Systems and Networks*, pages 103–112, 2002.

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HK00] Holger Hermanns and Joost-Pieter Katoen. Automated compositional Markov chain generation for a plain-old telephone system. *Science of Computer Programming*, 36(1):97–127, 2000.

[HKMKS99] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Keyser, and Markus Siegle. A tool for model checking markov chains. *Software Tools For Technology*, 1999.

[HKMKS00] H. Hermanns, J.P. Katoen, Joachim Meyer-Kayser, and Markus Siegle. A markov chain model checker. *Tools and Algorithms for the Construction and Analysis of Systems*, 1785:347–362, 2000.

[HKMKS01] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. Implementing a model checker for performability behaviour. In *Fifth Int. Workshop on Performability Modelling of Computer and Communication Systems*, 2001.

[HKNT98] G. Horton, V. Kulkarni, D. Nicol, and K. Trivedi. Fluid stochastic petri nets: Theory, application and solution techniques. *Europen Journal of Operational Research*, 105(1):184–201, 1998.

[HM96] H. Hermanns and V. Mertsiotakis. A stochastic process algebra based modelling tool. *Performance Engineering of Computer and Telecommunications Systems*, 1996.

[HN96] Boudewijn R. Haverkort and Ignas G. Niemegeers. Performability modelling tools and techniques. *Performance Evaluation*, 25(1):17–40, 1996.

[HT93] Boudewijn Haverkort and Kishor S. Travedi. Specification techniques for markov reward models. *Discrete-Event Dynamic Systems: Theory And Applications*, 3:219–247, 1993.

[HW02] L. He and J. Walrand. Dynamic provisioning of service level agreements between interconnected networks. In *Stochastic Networks 2002*, Stanford, CA, 2002.

[JHA02] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen A2. A probabilistic extension of uml statechartsi specification and verification. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Oldenburg, Germany, September 2002.

[KHMK00] W. Knottenbelt, P. Harrison, M. Mestern, and P. Kritzinger. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Performance Evaluation*, 2000.

[Kno96] William J. Knottenbelt. Generalised markovian analysis of timed transition systems. Master's thesis, Dept of Computer Science, University of Cape Town, 1996.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[KWA99] Justus Klingemann, Jurgen Wasch, and Karl Aberer. Deriving service models in cross-organizational workflows. In *Proceedings of the International Workshop on Reasearch Issues in Data Engineering*, pages 100–107, Sydney. Australia, 1999.

[Lib01] Information Technology Infrastructure Library. Service Management. Online: `http://www.itil.org/itil_e/itil_e_040.html`, 2001.

[LQV01] L. Lavazza, G. Quaroni, and M. Venturelli. Combining uml and formal notations for modelling real-time systems. In *Proceedings of the 8th European software enguineering conference*, Vienna, Austria, 2001. ACM Press.

[MA98] Daniel A. Menasc and Virgilio A. F. Almeida. *A Step-By-Step Approach To Capacity Planning In Client/Server Systems*, chapter 5. Prentice Hall, 1998.

[MC01] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing uml with formal languages. In *Proceedings of the 23rd international conference on Software engineering*, Toronto, Canada, 2001. IEEE Computer Society.

[MCT94] J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability*, 1(2):9–20, 1994.

[Mic03] Sun Microsystems. Java Standard Edition, Version 1.4.2.
Online: `http://java.sun.com/j2se/1.4.2/docs/api/`, 2003.

[QS94] Muhammad A. Qureshi and William H. Sanders. Reward model solution methods with impulse and rate rewards: An algorithm and numerical results. *Performance Evaluation*, 20(4):413–436, 1994.

[QS96] Muhammad A. Qureshi and William H. Sanders. A new methodology for calculating distributions of reward accumulated during a finite interval. In *Symposium on Fault-Tolerant Computing*, pages 116–125, 1996.

[RT99] Sandor Racz and Miklos Telek. Performability analysis of markov reward models with rate and impulse reward. In *Proceedings of the International Conference on Numerical Solutions to Markov Chains*, pages 169–187, 1999.

[Som96] Ian Sommerville. *Software Engineering*. Addison Wesley, 1996.

[SSB+95] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[ST88] R. M. Smith and Kishor S. Trivedi. Performability analysis: Measures, an algorithn and a case study. *IEEE Transactions On Computers*, 37(4):406–417, 1988.

[Tay82] John R. Taylor. *An Introduction to Error Analysis*. University Science Books, 1982.

[TH00] Dave Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[Wal45] Abraham Wald. Sequential testing of statistical hypothesis. *Annals of Mathematical Statistics*, 16(2), June 1945.

[YKNP04] Håkan L. S. Younes, Marta Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking: An empirical study. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Barcelona, Spain, March 2004. Springer.

[YMS03] Håkan L. S. Younes, David J. Musliner, and Reid G. Simmons. A framework for planning in continuous-time stochastic domains. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, pages 195–204, Trento, Italy, June 2003. AAAI Press.

[You03] Håkan L.S. Younes. Extending pddl to model stochastic decision processes. In *Proceedings of the ICAPS-03 Workshop on PDDL*, pages 95–103, Trento,Italy, June 2003.

[YS02] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235, Copenhagen, Denmark, July 2002. Springer.

Appendix A

mrmGen Specification Files

A.1 Processing

```
State Space
dimension:
[[(1..1000)],[0]]
:dimension

Labels allowd in model
labeldeclaration:
failure,processing,hdfailure,netfailure
:labeldeclaration

Processing/Failure label
label:
if reward == 0
  "failure"
else
  "processing"
end
:label

Hard Drive Failure label
label:
if reward < (1000*0.25)
```

```
  "hdfailure"
end
:label

It takes approximately a single node 4 hours
to process an event hence this reward
reward:
return coord[0]*0.25
:reward

Hard Drive failure rate
transition:
if from[0] == to[0] + 1
  from[0] * 0.000022831
else
  nil
end
:transition

Hard Drive repair rate
transition:
if from[0] + 1 == to[0] and from[0] > 0
  0.083333333
else
  nil
end
:transition

Total Power failure rate
transition:
if from[0] > 0 and to[0] == 0
  0.000342466
else
  nil
end
:transition
```

```
Total Power repair rate
transition:
if from[0] == 0 and to[0] == 250
  0.041666667
else
  nil
end
:transition
```

A.2 Cost

```
dimension:
[[(1..1000)],[0]]
:dimension

labeldeclaration:
finished,processing,hdfailure
:labeldeclaration

label:
if coord[0] == 0
  "finished"
else
  "processing"
end
:label

label:
if reward > 0 and coord[0] != 0
  "hdfailure"
end
:label

reward:
if coord[0] == 1000 or coord[0] == 0
  0.01
else
```

```
  70
end
:reward

HD Failure
transition:
if from[0] == to[0] + 1
  from[0] * 0.000022831
else
  nil
end
:transition

HD Repair
transition:
if from[0] + 1 == to[0] and from[0] > 0
  0.083333333
else
  nil
end
:transition

Finished Processing After A Year
transition:
if from[0] != 0 and to[0] == 0
  0.000114155
else
  nil
end
:transition
```