Imperial College London

Department of Computing

MEng Final Report

Automatic construction of product-form solutions in stochastic networks

Author:
Rhea Potdar

Supervisor:
Prof. Peter Harrison

Second Marker:
Dr. Giuliano Casale

Abstract

The growing complexity of modern systems has escalated the need for performance metrics. Stochastic models such as queuing networks and stochastic Petri nets have been used to model these systems so that their performance measures can be evaluated analytically. Product-form solutions are equilibrium state probabilities in networks of stochastic nodes (e.g. queues) in the form of a product of terms relating to each node separately. One can derive various performance metrics from these product form solutions, thus there is considerable effort dedicated to finding them in various stochastic models.

An established theorem, the Reversed Compound Agent Theorem (RCAT), derives mechanically the product-form solutions for stochastic models defined as a composition of two or more smaller stochastic models, under some conditions. Its use of the divide-and-conquer approach solves problems of state space explosion and computational complexity, which standard methods face while finding product-forms for large and complex networks.

This report presents a working implementation of RCAT in MATLAB and its extension Multiple Agent RCAT which can be applied to a wide variety of queuing networks with multiple components. It also provides the first working implementation of RCAT applied to stochastic Petri nets thus expanding its utility to analyse models composed of both Petri nets and queuing networks.

Acknowledgements

I would like to thank my supervisor, Prof. Peter Harrison, for his tremendous support, motivation and invaluable advice throughout this project, and also my family and friends for their encouragement and blessings.

Contents

Chapter 1

Introduction

1.1 Motivation

In the recent years, there has been an increase in the complexity of computer systems making performance models essential for understanding the behaviour of these systems. These performance models help determine performance measures of computer systems such as utilisation, throughput and help ensure that the systems can manage varying workload, that the utilisation of resources is fair and the list goes on. To analyse these computer systems, they are described abstractly using stochastic models which are then used to evaluate various performance measures of the systems.

Queuing networks (models with underlying Markov Processes) and stochastic Petri nets are two common stochastic models which are known to accurately describe computer systems and can be evaluated analytically. However due to the increase in complexity, the number of components of systems has increased, leading to a large state space and thus very high computational costs while analysing stochastic models. To improve the efficiency of this process, much effort has been devoted to finding the *product-form* solutions for the steady state probabilities of systems. The product-form solution, when it exists, can be derived for the joint steady state probabilities of interacting Markov processes by finding the the reversed process of the interaction. Analytically, this involves solving Kolmogorov (balance) equations [1] which can quickly cause state space explosion making the process computationally prohibitive for systems composed of many components.

To ease the computational difficulty, the Reversed Compound Agent Theorem (RCAT) [3] has been proved, which provides a method for finding product-form solutions using the divide-and-conquer approach. RCAT derives mechanically the product form solutions for steady state probabilities of stochastic models defined as a cooperation (or synchronisation) of two or more smaller stochastic models under some conditions. This approach for deriving the steady state probabilities of Markov processes does not require Kolmogorov equations to be solved. RCAT uses PEPA (Performance Evaluation Process Algebra), a Markovian Process Algebra formalism, which has an appropriate recursive structure for hierarchical analysis done by RCAT. But this does not limit the application of RCAT to systems specified by Markovian Process Algebra. It

can also be applied to derive product-forms in Stochastic Petri Nets which are specified diagrammatically and are more difficult to trace.

The RCAT theorem is proven to automatically derive product form solutions of G-networks [7] with negative customers, with ease comparable to deriving product form solutions of simpler Jackson queuing networks. This shows its compositional utility in validating product form solutions [3]. It can also be used for finding new product form solutions of networks with no known product form - such as blocking networks.

1.2 Contributions

The main contributions of this project are summarised below:

- Automatic construction of 'rate equations' used to derive product form solutions by implementing the RCAT and the Multiple Agent RCAT [9], allowing construction of product forms of queuing models composed of more than two processes.
- Automatic construction of 'rate equations' used to derive product form solutions of Stochastic Petri Nets (SPNs).
- A parser which translates a pure PEPA description as text input into a format required in the RCAT implementation. It provides a validator which checks to see whether the textual input actually resembles a meaningful PEPA process.
- A formalism for specifying a hierarchically defined subset of SPNs as programmable input and a parser for translating that input into a format required by the implementation.
- Ensuring the accuracy of the results generated by running the implementation over a varied class of queuing models and Petri nets and ensuring the implementation has ease of use by providing a clean and simple API.

1.3 Report Structure

The remainder of the report is organised as follows:

- **Chapter 2** provides a brief introduction to Markov Chains, Markov processes, Reversed Processes and Queuing networks. It then covers the theory necessary to comprehend the RCAT such as PEPA and proceeds to explain RCAT and (E)RCAT, and concludes with a background to Stochastic Petri Nets and product form Building blocks.
- **Chapter 3** describes the design choices and the implementation details of automating RCAT, starting from the parser - PEPA to MATLAB - and concluding with generating a system of rate equations and showing how to use them to generate product form solutions.

- **Chapter 4** describes the implementation details of extending RCAT to MARCAT and the implementation details of implementing RCAT for SPNs. It details parsing choices, a new formalism for the SPNs, shows how product form solutions are generated for SPNs and concludes with a design analysis for implementing RCAT for chains of interactions between queues.
- **Chapter 5** evaluates the project by running the implementation against a variety of queuing models and Petri nets and reviews the limitations and overall contributions of the project.
- **Chapter 6** summarises the contributions of this project as a formal conclusion to the report and includes suggestions for extending the project in the future.

Chapter 2

Background

In this chapter, we will explore the theory required for the automatic generation of product forms by implementing the RCAT. We commence this chapter by describing the relavent theory behind Markov Chains, Markov Processes and Reversed Process as shown in [5, 1, 2].

2.1 Markov Chains

In order to comprehend Markov Chains and Markov Processes, one must be familiar with stochastic processes.

Definition 2.1.1
A **stochastic process S** is defined as a family of random variables $\{X_t \in \Omega \mid t \in T\}$, which take values from some sample space Ω and are indexed by values from some parameter space T. Ω and T may be either discrete or continuous.

A **Markov chain** is a stochastic process that has the *Markov Property* with a countable sample space (or state space) Ω.

Definition 2.1.2
The **Markov Property (MP)** states that

$$P(X_{t+s} = j \mid X_u,\ u \leq t) \quad = \quad P(X_{t+s} = j \mid X_t) \tag{2.1}$$

This states that the conditional probability distribution of future states depends only on the current state and not the events that preceded it.

Markov chains can also be denoted as labelled transition systems, undergoing transitions with different rates between finite number of possible states.

2.1.1 Discrete Time Markov Chains

Discrete Time Markov Chains (DTMC) are Markov chains with a parameter space T consisting of discrete times $\{t_0, t_1, ...\}$. We are interested in the behaviour of a DTMC at equilibrium and the following results are used in defining it.

Definition 2.1.3
The m-step transition probabilities of a Markov chain defined as

$$\begin{aligned} p_{ij}^{(m)} &= P(X_{n+m} = j \mid X_n = i),\ (m \geq 1) \\ &= (P^m)_{ij} \end{aligned}$$

Since we are interested in the long term behaviour of DTMC, we can use its m-step transition probabilities to calculate the probabilistic behaviour of DTMC over any finite period of time. Thus the probability of a DTMC being in an arbitrary state j at equilibrium is defined as:

$$\begin{aligned} \pi_j &= lim_{n\to\infty} P(X_n = j \mid X_0 = i) \\ &= lim_{n\to\infty} (P^n)_{ij} \end{aligned}$$

Definition 2.1.4
If C is a subset of states, then it is called **closed** if j $\notin$ C implies j cannot be reached from any i $\in$ C .

If $\nexists$ a proper subset $C \subset \Omega$ which is closed, then the Markov chain is called *irreducible*.

Definition 2.1.6
The state j is **periodic** with period $m > 1$ if

$$p_{ii}^{(k)} = 0 \quad , \quad k \neq rm \text{ for any } r \geq 1$$

and

$$P(X_{n+rm} = j \text{ for some } r \geq 1 \mid X_n = j) \quad = \quad 1$$

Otherwise the state is *aperiodic*, or has period 1. An aperiodic DTMC is one in which all states are aperiodic.

Definition 2.1.5
Let m_j is the mean interval between successive visits to state j. If $m_j < \infty$ and $\pi_j = 1/m_j$, then $\pi_j > 0$ and state j is recurrent non-null or **positive recurrent**.

A positive recurrent DTMC is a DTMC in which all states are positive recurrent.

Proposition 2.1.1
If $\{X_n \mid n = 0, 1, ...\}$ is an irreducible, aperiodic Markov chain, then the limiting probabilities $\{\pi_j | j = 0, 1, ...\}$ exist and $\pi_j = 1/m_j$ where m_j is the mean interval between successive visits to state j.

When the limiting probabilities $\{\pi_j | j = 0, 1, ...\}$ do exist, they form the steady state probability distribution (SSPD) of a DTMC. This is formally defined by the following Theorem 2.1.1.

Theorem 2.1.1
An irreducible, aperiodic Markov Chain, X, with state space S and one-step transition probability matrix $P = (p_{ij} \mid i, j \in S)$, is positive recurrent if and only if the system of equations

$$\pi_j = \sum_{i \in S} \pi_i p_{ij}$$

and (normalisation):

$$\sum_{i \in S} \pi_i = 1$$

has a solution. If it exists, the solution is unique and is the SSPD of X.

2.2 Markov Processes

A Markov process(MP) is a stochastic process, which has a *continuous* parameter space T, discrete sample space Ω and the *Markov property* (refer equation 2.1). They can also be defined as Markov chains with continuous time parameters.

A Markov process is *time homogenous* if the transition probability function of MP, $p_{ij}(s) = P(X_{t+s} = j \mid X_t = i)$, is independent of t, or equivalently $p_{ij}(s) = P(X_s = j \mid X_0 = i)$. Markov Property and time homogeneity imply the *memoryless property.*

Definition 2.2.1
The **memoryless property** of a Markov process states that if at time t the process is in state j, the time remaining in state j is independent of the time already spent in state j.

Using time homogeneity, the generators q_{ij} of a Markov Process can be uniquely determined by the products:

$$q_{ij} \quad = \quad \mu_i p_{ij}$$

where μ_i is the rate out of state i and p_{ij} is the probability of selecting state j next. q_{ij} is also the instantaneous transition rate from state i to state j, $i \neq j$. They also form $\boldsymbol{Q}$, generator matrix of the Markov Process, in which all rows sum to zero by setting $q_{ii} = -\mu_i$. So $Q = (q_{ij})$.

2.2.1 SSPD of the Markov Process

If a Markov Process is positive recurrent, the limits π_j exist, then $\pi_j > 0, \sum_{j \in S} \pi_j = 1$ and $\{\pi_j \mid j \in S\}$ constitute the SSPD or Steady State Probability Distribution of the Markov Process. This is formally defined by Theorem 2.2.1.

Theorem 2.2.1
An irreducible Markov Process X with state space S and generator matrix $Q = (q_{ij})$ $(i, j \in S)$ is positive recurrent if and only if
$\forall j \in S$, Balance equations:

$$\sum_{i \in S} \pi_i q_{ij} \quad = \quad 0$$

and Normalising equation:

$$\sum_{i \in S} \pi_i \quad = \quad 1 \tag{2.2}$$

have a solution. This solution is unique and is the SSPD.

From Theorem 2.2.1, we can rewrite the balance equations as

$$\sum_{j \neq i} \pi_i q_{ij} \quad = \quad \sum_{j \neq i} \pi_j q_{ji}$$

Following gives the justification of the balance equations:

In equilibrium, π_i is the proportion of time that the process spends in state i and q_{ij} is the rate at which the process goes from state $i \to j$ $(j \neq i)$. Thus, in unit time, the expected number of transitions from state i to state j is $\pi_i q_{ij}$. This quantity is called the *probability flux* from state i to j. So we can infer that the left-hand side of the balance equation for state i is the total flux out of state i to any other state. Similarly, the right-hand side is the total flux into state i from any other state.

$\implies \forall j$, the fluxes balance:

$$\sum_{i \neq j} flux(i \to j) \quad = \quad \sum_{i \neq j} flux(j \to i) \tag{2.3}$$

2.2.2 Example - Poisson Process

The Poisson process is a renewal process with renewal period (inter-arrival time) having cumulative distribution function F and probability density function (pdf) f

$$\begin{aligned} F(x) &= P(X \leq x) = 1 - e^{-\lambda x} \\ f(x) &= F'(x) = \lambda e^{-\lambda x} \end{aligned}$$

where λ is the rate of the Poisson process. Since its an example of a Markov process its probability of arrival in period $(t, t+h)$ is independent of the process history before t. So by memoryless property:

$$\begin{aligned} P(\text{arrival in } (t, t+h)) &= 1 - e^{-\lambda h} \\ &= \lambda h + o(h) \end{aligned}$$

From the above result we get the instantaneous transition rates which can be then used to find the SSPD for the process.

$$q_{ij} \quad = \quad \begin{cases} \lambda & \text{if } j = i+1 \\ 0 & \text{if} j \neq i, i+1 \\ & \text{not defined if } j = i \end{cases}$$

2.2.3 Birth-Death Processes

Birth-death process is a special case of Markov Process with state space $\{0, 1, ...\}$ in which a one-step transition can only change the current state by one unit, so if $i \to j$ then $|i - j| = 1$. This process thus has only non-zero transition probabilities - $a_{i,i+1}$ and $a_{i+1,i}$ $(i \geq 0)$, representing *births* and *deaths* respectively. This ensures the population need not become extinct when state 0 is reached which is useful while considering queues, where arrivals can join an empty queue, represented by state 0. The SSPD of this process is discussed later taking M/M/1 queue as an example.

2.3 Single Server Queue (SSQ)

The Single Server Queue Model [1] - SSQ - is a birth-death process that consists of

- a Poisson arrival process with a rate of λ
- a queue which the arriving tasks join
- a server with a FIFO queuing discipline and exponentially distributed service times with parameter μ

The M/M/1 queue is an example of the SSQ model.

2.3.1 Kendall's Notation

Queues are classified according to Kendall's notation [1, 5], which defines the class $A/S/m/K/N/D$ as:

- A describes the nature of the arrival process. For example if the process is Poisson, then $A = M$ for Markovian.
- S describes the service time distribution. $S = M$ for a Markovian (exponential) service time distribution, while $S = G$ stands for a general or non-Markovian service time distribution.
- m denotes the number of servers available to give service to customers in the queue. $m = 1$ refers to a single server, while $m = m$ shows a parallel server.
- K denotes the capacity of the system or the maximum number of customers allowed in the system.
- N denotes the size of the population from which the customers come.
- D denotes the queuing discipline or priority order in which customers are served in the queue.

In this paper, the concise form - $\boldsymbol{A/S/m}$ - is used and default values $K = \infty,\ N = \infty,\ D = FIFO$ are assumed.

2.3.2 M/M/1 Queue

The M/M/1 queue [1] is an example of the SSQ model with a Poisson arrival process rate λ, Markovian service time distribution rate μ, unlimited server capacity and infinite calling population. The rates λ , μ are general functions of the queue length; so when the queue length is n, we write them as $\lambda(n)$,$\mu(n)$. Considering the M/M/1 queue in equilibrium, in the steady state, we can write down the probability flux balance equations passing in and out of the states shown in Figure 2.1.

There is only one outgoing arc and one incoming arc. The balance equations are therefore,

Outward flux (all from state i): $\pi_i\lambda(i)$, $\forall i \geq 0,\ i \in S$

Inward flux (all from state $i+1$): $\pi_{i+1}\mu(i+1)$, $\forall i \geq 0,\ i \in S$

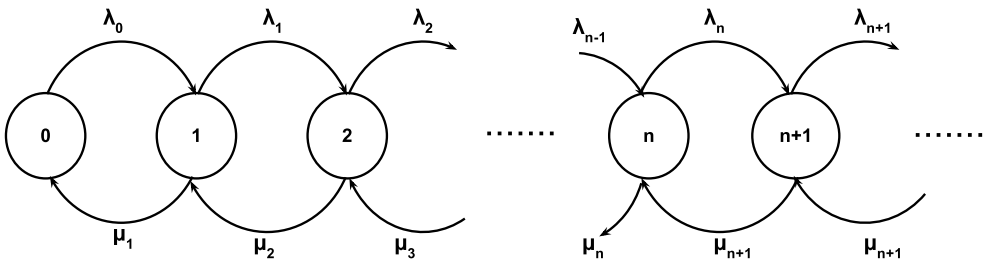

Figure 2.1: M/M/1 Queue state diagram

$\Longrightarrow$ balance equations (using equation 2.3)

$$\pi_i \lambda(i) \quad = \quad \pi_{i+1} \mu(i+1)$$

So,

$$\begin{aligned} \pi_{i+1} &= \frac{\lambda(i)}{\mu(i+1)} \pi_i \\ &= [\prod_{j=0}^{i} \rho(j)] \ \pi_0 \\ \text{where } \rho(j) &= \frac{\lambda(i)}{\mu(i+1)} \end{aligned}$$

The Normalising equation (refer equation 2.2) implies that

$$\pi_0 (1 + \sum_{i=0}^{\infty} \prod_{j=0}^{i} \rho(j)) \quad = \quad 1$$

Solving the equations we get the steady state probability π_i for any state $i \geq 0$:

$$\pi_i \quad = \quad \frac{\prod_{j=0}^{i-1} \rho(j)}{\sum_{k=0}^{\infty} \prod_{n=0}^{k-1} \rho(n)} \ (i \geq 0)$$

In a classical M/M/1 queue, the arrival and service rates, λ and μ respectively, are constant. So $\forall n \in S$, $\lambda(n) = \lambda$, $\mu(n) = \mu$, $\rho(n) = \rho = \frac{\lambda}{\mu}$. This implies,

$$\begin{aligned} \pi_i &= \frac{\prod_{j=0}^{i-1} \rho^j}{\sum_{k=0}^{\infty} \prod_{n=0}^{k-1} \rho^n} \\ &= (1-\rho)\rho^i \ (i \geq 0) \end{aligned} \tag{2.4}$$

The derived result (equation 2.4) is the simplified version of the SSPD of M/M/1 queues.

Since this (equation 2.4) is a geometric mass probability function, it is easy to deduce the mean length of the queue (L) and utilisation of the server (U) in equilibrium.

$$\begin{aligned} L &= \frac{\rho}{(1-\rho)} \\ U &= 1 - \pi_0 = \rho \end{aligned}$$

It can be noted here that, the mean arrival rate (λ) is equal to the mean departure rate ($U\mu$) in steady state, as required.

The above analysis and argument applies to any system in equilibrium.

2.4 Reversed Processes

A *reversed process* [6, 3, 1, 5] of a stationary Markov process is a stochastically identical process (to the original MP) with the same state space but in to which the direction of time has been reversed. Knowledge of the reversed process allows us solve balance equations of a Markov process in equilibrium and therefore obtain product-form stationary distributions of complex processes such queuing network models. Thus comprehension of reversed processes helps in applying the RCAT theorem (discussed in Section 2.6.2).

Definition 2.4.1
A stochastic process $\{X_t \mid -\infty < t < \infty\}$ is **stationary** if

$$(X_{t_1}, X_{t_2}, \ldots, X_{t_n}) \quad \text{and} \quad (X_{t_1+\tau}, X_{t_2+\tau}, \ldots, X_{t_n+\tau})$$

and have the same probability distribution for all times $t_1, t_2, \ldots, t_n$ and τ.

Definition 2.4.2
A stochastic process $\{X_t \mid -\infty < t < \infty\}$ is **reversible** if

$$(X_{t_1}, X_{t_2}, \ldots, X_{t_n}) \quad \text{and} \quad (X_{\tau-t_1}, X_{\tau-t_2}, \ldots, X_{\tau-t_n})$$

and have the same probability distribution for all times $t_1, t_2, \ldots, t_n$ and τ.

Definitions 4.1 and 4.2 relate a stationary Markov process to its reversed process. Thus the reversed process of a Markov process $\{X_t\}$ will be the stationary process $\{X_{\tau-t}\}$ for any real number τ. We can also define a reversed process in terms of balance conditions of a stationary Markov process as in the following Proposition 2.4.1.

Proposition 2.4.1
A stationary Markov process $\{X_t\}$ with a generator matrix $Q = (q_{ij})$ is reversible if and only if there exists a collection of positive real numbers $\{\pi_k \mid k \in S\}$ satisfying the detailed balance equations:

$$\pi_i q_{ij} = \pi_j q_{ji} \qquad (\forall i, j \in S, i \neq j)$$

An example of a reversible process is the M/M/1 queue. An M/M/1 queue is a birth-death process and thus its transition graph is linear - a tree with no branches. So, the probability flux in and out of states balances as derived at equation 2.3. Thus by Proposition 2.4.1 the M/M/1 queue is reversible.

The departure process of an M/M/1 queue is also identical to the arrival process in the reversed queue. To prove this claim let process N_t denote the number of customers in the queue at time t. An arrival corresponds to the instants N_t jumps up by one and defines a Poisson arrival process. Due to reversibility, instants at which N_{-t} jumps upwards by one also define a reversible process. But arrivals in N_{-t} become departures in N_t thus proving that the departure process forms an identical Poisson process.

Proposition 2.4.1 would be useful to detect a reversible Markov process but most Markov processes are not reversible. Thus a method is required to define a reversed process $\{X_{\tau-t}\}$ for a Markov process $\{X_t\}$ that is not reversible. The stationary distribution π is the same for both the processes and thus the we can relate the instantaneous transition rates of the reversed process to those of the original process by the following Proposition 2.4.2.

Proposition 2.4.2
The reversed process of a stationary Markov process X_t with state space S, generator matrix Q and stationary probabilities π is a stationary Markov process with generator matrix Q' defined by

$$q'_{ij} = \frac{\pi_j q_{ji}}{\pi_i} \qquad \forall i, j \in S \tag{2.5}$$

and with the same stationary probabilities π.

2.4.1 Kolmogorov's criteria

The equilibrium distribution of a stationary Markov process can be found using the result (Equation 2.5) from Proposition 2.4.2 by *guessing* possible instantaneous transition rates $\{q'_{ij} \mid i, j \in S\}$ for the reversed process and a collection of positive real numbers $\{\pi_i \mid i \in S\}$ which sum finitely to G such that

- The total rate out of state i is the same for reversed and original process: $q'_i = q_i$ $(\forall i \in S)$ where $q_i \equiv -q_{ii}$ is the total rate out of state i

- $\pi_i q'_{ij} \;=\; \pi_j q_{ji}$ $(\forall i \;,\; j \in S \;,\; i \neq j)$

These conditions ensure that π satisfies the Markov process balance equations and thus the steady state probabilities are $\{\pi_i/G \mid i \in S\}$ by uniqueness, where G is the normalizing constant.

But this methodology depends on making the 'right guesses' of the unknown vector π - the equilibrium state probabilities. Since π is defined by the generators of a Markov process, its instantaneous transition rates, another methodology which finds reversed processes without reference to π can be useful. The following proposition, called *Kolmogorov's criteria,* provides this by placing conditions only on the instantaneous rates of a Markov process.

Proposition 2.4.3 - Kolmogorov's Generalised Criteria
A stationary Markov process with state space S and generator matrix Q has a reversed process with generator matrix Q' if and only if

$$q'_i = q_i \qquad \forall i \in S$$

and for every finite sequence of states i_1, i_2, ..., $i_n \in S$,

$$q_{i_1 i_2} q_{i_2 i_3} \cdot \;.\; . \; q_{i_{n-1} i_n} q_{i_n i_1} \;=\; q'_{i_1 i_n} q'_{i_n i_{n-1}} \;.\; .\; . \; q'_{i_3 i_2} q'_{i_2 i_1} \tag{2.6}$$

where $q_i \;=\; -\,q_{ii} \;=\; \sum_{j \,:\, j \neq i} q_{ij}$ is the total exit rate from state i.

As mentioned, Proposition 2.4.3 can be used to find the instantaneous transition rates of the reversed process Q' and then use them to derive the SSPD of both original and reversed Markov Process. This can be done by using the Equation 2.5 or a modified approach given as follows:

1. We first arbitrarily choose a reference state 0
2. We then find a sequence of directly connected states 0, . . . , j in either the forward or reverse process
3. We then calculate the steady state probability π_j related to a base value π_0

$$\begin{aligned} \pi_j &= \pi_0 \prod_{i=0}^{j-1} \frac{q_{i,i+1}}{q'_{i+1,i}} \\ &= \pi_0 \prod_{i=0}^{j-1} \frac{q'_{i,i+1}}{q_{i+1,i}} \end{aligned}$$

2.4.1.1 Example

Consider a 3-state CTMC, as shown in Figure 2.2, with the only non-zero transition rates given by $q_{12} = q_{23} = \lambda$ and $q_{32} = q_{31} = \mu$. Thus using

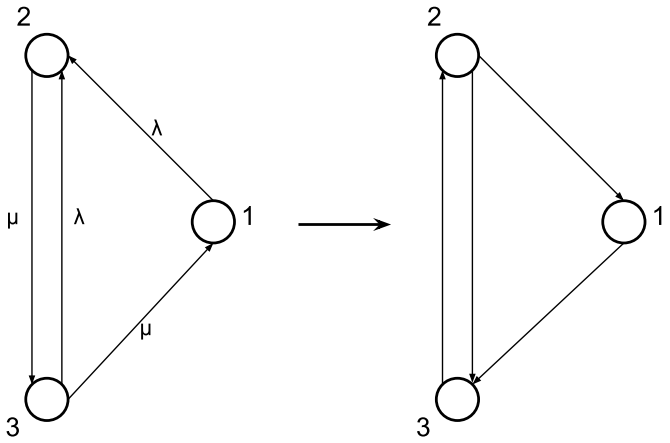

Figure 2.2: A forward Markov process and its reversed counterpart

Kolmogorov Generalised Criteria (Proposition 2.4.3) to discover the reversed rates in its generator matrix Q' , we start by comparing the outbound rates in the forward and reversed process:

$q_1 = \lambda$, $q_2 = \lambda$, $q_3 = 2\mu$ gives,

$$q'_1 = q'_{13} = q_1 = \lambda \ , \quad q'_2 = q'_{21} + q'_{23} = q_2 = \lambda \ , \quad q'_3 = q'_{32} = q_3 = 2\mu$$

There are two minimal cycles in the forward process, $\{1 \to 2,\ 2 \to 3,\ 3 \to 1\}$ and $\{2 \to 3,\ 3 \to 2\}$, which give the following cycle equations:

$$q'_{13}q'_{32}q'_{21} = \lambda^2\mu \qquad q'_{23}q'_{32} = \mu\lambda$$

Solving these we get,

$$q'_{13} = \lambda,\ q'_{32} = 2\mu,\ q'_{21} = q'_{23} = \frac{\lambda}{2}$$

This style of reasoning is used to prove the RCAT and to determine the reversed rates of PEPA actions in sequential PEPA components, described in later sections.

2.5 Queuing Networks

Queuing networks [1, 5] are systems that are network of queues with connected inputs and outputs. *Open* queuing networks are queuing networks where external customers are allowed to arrive or depart the system while *closed* queuing networks are those where no external customers are allowed in the system.

2.5.1 Traffic Equations

Traffic equations are a system of linear equations used to compute the mean arrival rate values λ_i to each node i in the network. In queuing networks, for any node i, the mean number of arrivals to node i is the sum of the mean external arrivals to node i and the mean arrivals from all other nodes j.

So the traffic equations for node $i = 1, 2, \ldots, M$ is defined as:

$$\lambda_i = \gamma_i + \sum_{j=1}^{M} \lambda_j q_{ji}$$

where γ_i is the external arrival rate at node i, and q_{ji} is the routing probability that traffic leaving node j is routed to node i. In closed queuing networks, the external arrival rate γ_i will be zero.

2.5.2 Jackson Theorem

Using the traffic equations, the utilisation of node i can be derived. For open queuing networks having nodes with fixed service rates μ_i, the traffic intensity is

$$\rho_i = \frac{\lambda_i}{\mu_i}$$

The utilisation therefore at node i is U_i = traffic intensity at node i = ρ_i. A network is *stable*, which means capable of reaching a steady state, if the following condition holds:

Utilisation at all nodes should be:

$$\rho_i = \frac{\lambda_i}{\mu_i} < 1 \qquad : \forall i$$

To calculate the steady state probability distribution of a stable network, the Jackson's theorem is used, which gives a product form solution for open queuing networks.

Theorem 2.5.1 Jackson Theorem
If the open queuing network is stable, that is if the condition:

$$\rho_i < 1 \qquad : \forall i$$

holds, then the steady-state exists and

$$\pi(n_1, \ldots, n_m) \;=\; \prod_{i=1}^{M} (1-\rho_i)\rho_i^{n_i} \tag{2.7}$$

is the joint probability (or steady state probabilities) where
$\pi(n_1, \ldots, n_m)$ is the probability that the system has queue length n_i at node i . The result is also called the *product form solution* of the model.

The product-form result implies that each node can be reasoned about as a M/M/1 queue in isolation and thus makes obtaining performance measures of a network such as - mean queue lengths, throughput, mean waiting time - an easy task. If we analyse the product form solution, we notice that the marginal distribution of the number of jobs at node i is the same as that of an M/M/1 queue (compare the SSPD solution of a M/M/1 queue from 2.4 to the inner term of the product form solution obtained 2.7). Thus we can conclude that for a stable Jackson Network with an arrival rate λ_i to node i:

- The number of of jobs at any node is independent of the state of any other node as we get a product form solution
- Node i behaves stochastically as if it were subject to Poisson arrivals with rate λ_i.

We can now define (an open) *Jackson Network[1]* as an open queuing network with any external arrivals to any node i forming a Poisson stream and the equilibrium probability distribution resulting in to a product form solution model. It is a relatively simple queuing network and we shall derive its product form solution using the RCAT theorem in later sections.

2.5.3 G-Network

A G-network [7], also known as a generalised queueing network or Gelenbe network and introduced by Erol Gelenbe, are queuing networks with negative customers. Thus this network has two types of customers:

- *positive customers* are customers as in a M/M/1 queue which arrive from other queues or externally as Poisson arrivals. Their departures (state decrements) are synchronised with state increments (arrivals) in the destination queue.

- *negative customers* are those customers which remove or 'cancel' positive customers in the queue if it is not empty and have no effect on an empty queue. They are useful to remove traffic if a network is congested.

A product form solution exists for stationary G-networks despite the traffic flows forming a system of non-linear equations.

2.6 Formalisms and product-forms

2.6.1 PEPA

PEPA[8, 3, 6, 4] is a formal system description language used in performance modelling [8]. It is a Markovian Process Algebra (MPA) with the fewest combinators necessary to provide a semantic model for denoting the states of a continuous time Markov Chain [4]. As the Jackson Theorem (see Theorem 2.5.1) operates over queuing networks to generate product form solutions, RCAT operates over PEPA.

In PEPA, a system is an interaction of *components* which engage in *activities.* For example, in a stochastic process components correspond to states while activities correspond to transitions between them. An activity in PEPA is specified as $\alpha = (a, r)$, where a is the *action type* and r is the *activity rate.* The Markov process's transition rates are represented in PEPA by the activity rate as a duration which is a random variable with an exponential distribution. Every activity within the PEPA model with the same action type represents different instances of that action in the system. A *derivation graph*, formed by PEPA terms at nodes (states of an MP) and arcs showing transitions between them, determine the underlying Markov process of a component P.

2.6.1.1 PEPA Syntax

Definition 2.6.1
The syntax[6] of a PEPA component P is represented by:

$$P \quad ::= \quad (a, \lambda).P \mid P_1 \; + \; P_2 \mid P_1 \underset{\mathcal{L}}{\bowtie} P_2 \mid P/L \mid A$$

$(a, \lambda).P$ is called a *prefix* operation. It represents a process which performs an action a with a rate parameter λ and then becomes a new process P. The rate parameter may either be a positive real number or the value $\top$ which makes the action passive in a cooperation.

$P_1 + P_2$ is a *choice* operation. Here the two components P_1 and P_2 are in a race condition where the process can evolve into either one of them. The first component to activate will dictate the direction of choice

$P_1 \underset{\mathcal{L}}{\bowtie} P_2$ is the *cooperation* or synchronisation operation. P_1 and P_2 run in parallel and synchronise over actions in set $\mathcal{L}$. Synchronising actions must be

activated jointly by both components. Thus if component P_1 can evolve only by activating a synchronising action a, it may be blocked until P_2 is in a derivative state that can synchronise on a. In a passive cooperation, if P_1 evolves with a rate $\top$ on synchronising action a, then the joint action a inherits its rate from the P_2 component alone. Parallelism is a special case of synchronisation where the set of synchronising actions is empty, that is, $P_1 \underset{\emptyset}{\bowtie} P_2$.

P/L is the *hiding* operation. Observable actions from set L in P are rewritten as silent τ actions which cannot be used in cooperations with other components.

A is a *constant* label that is used while constructing recursive definitions.

Processes or *agents* defined using only assignments or prefixes are called simple agents while the ones defined using at least one cooperation combinator are called compound agents.

2.6.1.2 PEPA activity substitution

Relabelling[6] or activity substitution is a method for an activity $\alpha = (a, r)$ to be syntactically replaced with activity $\alpha' = (a' , r')$. This is particularly useful in defining reversed processes of cooperations.

Definition 2.6.2
The PEPA activity substitution function is defined as:

$$
\begin{aligned}
(\beta.P)\{\alpha \leftarrow \alpha'\} &= \begin{cases} \alpha'.(P\{\alpha \leftarrow \alpha'\}) & : \text{ if } \alpha = \beta \\ \beta.(P\{\alpha \leftarrow \alpha'\}) & : \textit{otherwise} \end{cases} \\
(P + Q)\{\alpha \leftarrow \alpha'\} &= P\{\alpha \leftarrow \alpha'\} + Q\{\alpha \leftarrow \alpha'\} \\
\\
(P_1 \underset{\mathcal{L}}{\bowtie} P_2)\{\alpha \leftarrow \alpha'\} &= P\{\alpha \leftarrow \alpha'\} \underset{\mathcal{L}\{\alpha \leftarrow \alpha'\}}{\bowtie} Q\{\alpha \leftarrow \alpha'\} \\
\textit{where } \mathcal{L}\{(a, \lambda) \leftarrow (a', \lambda')\} &= \begin{cases} (\mathcal{L} \backslash \{a\}) \bigcup \{a'\} & : \textit{if } a \in \mathcal{L} \\ \mathcal{L} & : \textit{otherwise} \end{cases}
\end{aligned}
$$

2.6.1.3 Reversing a PEPA component

Reversing a PEPA component is done for finding the reversed process of a Markov process in terms of a PEPA agent with appropriate rates. The RCAT theorem deals with the reversal of a compound agent, $P_1 \underset{\mathcal{L}}{\bowtie} P_2$, and uses reversing sequential components in its definition.

Definition 2.6.3
For all states S in a sequential component:

$$\overline{S} \stackrel{def}{=} \sum_{i \,:\, R_i \rightarrow^{(a_i,\lambda_i)} S} (\overline{a}_i \, , \, \overline{\lambda_i}).\overline{R_i}$$

Thus the reversed rate of a component $\bar{S}$ is a choice between all the states that have S as its immediate successor in the forward process [6]. Thus in the reversed component, actions a become $\bar{a}$ and rates λ become $\bar{\lambda}$ where $\bar{\lambda}$ can be calculated using Kolmogorov's generalised criteria (Proposition 2.4.3 - Equation 2.6 for example). Finding the rates of a reversed compound agent requires a new rule as an agent may have several actions leading to the same state that synchronise with distinct actions in a cooperating agent [3].

Definition 2.6.4
The reversed actions of multiple actions (a_i, λ_i), for $1 \leq i \leq n$ that an agent P can perform, which lead to the same derivative Q, are respectively

$$(\bar{a_i}, \, (\lambda_i/\lambda)\bar{\lambda}))$$

where $\lambda = \lambda_1 + \, . \, . \, . \, + \lambda_n$ and $\bar{\lambda}$ is the reversed rate of the one-step, composite transition with rate λ in the Markov chain, corresponding to all the arcs between P and Q.

Definition 2.6.4 is used in the RCAT theorem for reversing compound agents.

2.6.2 Reversed Compound Agent Theorem (RCAT)

The Reversed Compound Agent Theorem (RCAT) finds the reversed compound agent of the cooperation $P \underset{L}{\bowtie} Q$ by finding the reversed rates of the constituent processes P and Q. For RCAT operation, we define some restrictions on actions in a component.

Definition 2.6.5
The subset of action types in a set L which are passive with respect to a process P (i.e. are of the form $(a, \top)$ in P) is denoted by $\mathcal{P}_P(L)$. The set of corresponding active action types is denoted by $\mathcal{A}_P(L) = L \backslash \mathcal{P}_P(L)$.

Thus an action cannot be both passive and active in the same component. If an action is active in a component, all its instances are active in that component, and if it is passive then all its instances are passive. This is necessary as we, in RCAT, syntactically transform every passive action before reversing an agent to ensure every passive action rate is uniquely identified with all instances of its action type [3].

2.6.2.1 The RCAT theorem

The RCAT as stated in the original paper [3]:

Theorem 2.6.1 Reversed Compound Agent Theorem
Suppose that the cooperation $P \underset{L}{\bowtie} Q$ has a derivation graph with an irreducible subgraph G. Given that:

1. every passive action type in $\mathcal{P}_P(L)$ or $\mathcal{P}_Q(L)$ is always enabled in P or Q respectively (i.e. enabled in all states of the transition graph);
2. every reversed action of an active action type in $\mathcal{A}_P(L)$ or $\mathcal{A}_Q(L)$ is always enabled in $\overline{P}$ or $\overline{Q}$ respectively;
3. every occurrence of a reversed action of an active action type in $\mathcal{A}_P(L)$ or $\mathcal{A}_Q(L)$ has the same rate in $\overline{P}$ or $\overline{Q}$ respectively.

the reversed agent $\overline{P \underset{L}{\bowtie} Q}$, with derivation graph containing the reversed subgraph $\overline{G}$, is:

$$R^* \underset{\overline{L}}{\bowtie} S^*$$

where:

$$\begin{aligned} R^* &= \overline{R}\{(\overline{a}, \bar{p}_a) \leftarrow (\bar{a}, \top) \mid a \in \mathcal{A}_P(L)\} \\ S^* &= \overline{S}\{(\overline{a}, \bar{q}_a) \leftarrow (\bar{a}, \top) \mid a \in \mathcal{A}_Q(L)\} \\ R &= P\{(a, \top) \leftarrow (a, x_a) \mid a \in \mathcal{P}_P(L)\} \\ S &= Q\{(a, \top) \leftarrow (a, x_a) \mid a \in \mathcal{P}_Q(L)\} \end{aligned}$$

where the symbolic rates $\{x_a\}$ are given by:

$$x_a = \begin{cases} \bar{q}_a & : \text{ if } a \in \mathcal{P}_P(L) \\ \bar{p}_a & : \text{ if } a \in \mathcal{P}_Q(L) \end{cases}$$

and $\bar{p}_a$ and $\bar{q}_a$ are symbolic rates of action types $\bar{a}$ in $\overline{P}$ and $\overline{Q}$ respectively.

The proof of this theorem is detailed in the paper [3] and consists of verifying that Kolmogorov's criteria hold.

2.6.2.2 (E)RCAT - Extended RCAT

Conditions 1 and 2 in the RCAT (Theorem 2.6.1) requires every passive action to be enabled in every derivative (state) of both the forward and reversed cooperating agents. This ensures that the total outgoing rate from any state is the same in the two processes in agent $P \underset{L}{\bowtie} Q$ and its reversed agent. But as stated in paper [10], relaxing these conditions allows RCAT to be applied on a large breadth of systems. We define some new notation to account for the

action types in L that might not be present in every derivative of the forward and reversed cooperating agents. This is an extension to Definition 2.6.5

Definition 2.6.6
$\mathcal{P}_A^{i\rightarrow}$ denotes the subset that are passive in A and correspond to transitions out of state i in the Markov process A ;
$\mathcal{P}_A^{i\leftarrow}$ denotes the subset that are passive in A and correspond to transitions into state i in the Markov process A ;
$\mathcal{A}_A^{i\rightarrow}$ denotes the subset that are active in A and correspond to transitions out of state i in the Markov process A ;
$\mathcal{A}_A^{i\rightarrow}$ denotes the subset that are active in A and correspond to transitions into state i in the Markov process A;
$\mathcal{P}^{(i,j)\rightarrow} = \mathcal{P}_P^{i\rightarrow} + \mathcal{P}_Q^{j\rightarrow}$ and $\mathcal{A}^{(i,j)\rightarrow} = \mathcal{A}_P^{i\rightarrow} + \mathcal{A}_Q^{j\rightarrow}$;
$\mathcal{P}^{(i,j)\leftarrow} = \mathcal{P}_P^{i\leftarrow} + \mathcal{P}_Q^{j\leftarrow}$ and $\mathcal{A}^{(i,j)\leftarrow} = \mathcal{A}_P^{i\leftarrow} + \mathcal{A}_Q^{j\leftarrow}$;
$\alpha_a^{(i,j)}$ denotes the instantaneous transition rate out of (joint) state (i,j) in the Markov process of $P \underset{L}{\bowtie} Q$ corresponding to active action type $a \in L$;
$\overline{\beta_a^{(i,j)}}$ denotes the instantaneous transition rate out of (joint) state (i,j) in the reversed Markov process of $P \underset{L}{\bowtie} Q$ corresponding to passive action type $a \in L$.

The theorem with the new notation is defined as follows:

Theorem 2.6.2 Extended Reversed Compound Agent Theorem*[9]*
If the following conditions hold,

1. The reversed rate x_a of every active action a is the same at every instance, given by the solution of the rate equations, as in the original RCAT (Theorem 2.6.1).

2. The forward and reversed passive and active transition rates satisfy:

$$\sum_{a \in \mathcal{P}^{(i,j)\rightarrow}} x_a - \sum_{a \in \mathcal{A}^{(i,j)\leftarrow}} x_a = \sum_{a \in \mathcal{P}^{(i,j)\leftarrow} \setminus \mathcal{A}^{(i,j)\leftarrow}} \overline{\beta_a^{(i,j)}} - \sum_{a \in \mathcal{A}^{(i,j)\rightarrow} \setminus \mathcal{P}^{(i,j)\rightarrow}} \alpha_a^{(i,j)}$$

Then the reversed process of the cooperation $P \underset{L}{\bowtie} Q$ is

$$\overline{P \underset{L}{\bowtie} Q} = R^* \underset{\overline{L}}{\bowtie} S^*$$

where:

$$\begin{aligned} R^* &= \overline{R}\{(\overline{a}, \bar{p}_a) \leftarrow (\bar{a}, \top) \mid a \in \mathcal{A}_P(L)\} \\ S^* &= \overline{S}\{(\overline{a}, \bar{q}_a) \leftarrow (\bar{a}, \top) \mid a \in \mathcal{A}_Q(L)\} \\ R &= P\{(a, \top) \leftarrow (a, x_a) \mid a \in \mathcal{P}_P(L)\} \\ S &= Q\{(a, \top) \leftarrow (a, x_a) \mid a \in \mathcal{P}_Q(L)\} \end{aligned}$$

where the symbolic rates $\{x_a\}$ are given by:

$$x_a = \begin{cases} \bar{q}_a & : \text{ if } a \in \mathcal{P}_P(L) \\ \bar{p}_a & : \text{ if } a \in \mathcal{P}_Q(L) \end{cases}$$

and $\bar{p}_a$ and $\bar{q}_a$ are symbolic rates of action types $\bar{a}$ in $\overline{P}$ and $\overline{Q}$ respectively.

2.6.2.3 Practical Application of the RCAT method

For using the RCAT method practically, the algorithm detailed below can be used. This algorithm does not require the whole reversed processes to be determined in RCAT Theorem 2.6.1 but does require the specific reversed rates of the synchronising active actions. These rates are computed using equation 2.5 which use the equilibrium state probabilities of each component process.

Generic Algorithm [9]

Consider the cooperation $P_1 \underset{L}{\bowtie} P_2$. The algorithm is as follows

1. From P_k construct R_k by setting the rate of every instance of action $a \in L$ that is passive in P_k to x_a, for $k = 1, 2$ (each a will be passive for only one k);

2. For each active action type a in R_k , $k = 1, 2$, check that its reversed rate is the same for all of its instances, that is for for all transitions $i \to j$ it denotes states i, j in the state transition graph of R_k. Compute and denote this reversed rate (in the reversed process $\overline{R_k}$) by the equation

$$\overline{r_a^i} \quad = \quad \frac{\pi_k(i) r_a^i}{\pi_k(j)} \tag{2.8}$$

where r_a^i is the specified forward rate (any, if more than one) of the instance of action type a going out of state i. In fact, if the reversed process of the cooperation is required, the full reversed processes R_k must be computed;

3. Noting that the symbolic reversed rate $\overline{r_a}$ will in general be a function of the $x_b (b \in L)$, solve the equations $x_a = \overline{r_a}$ for each $a \in L$ and substitute the solutions for the variables x_a in each R_k;

4. Check the enabling conditions (detailed in [12]) for each co-operating action in each process P_k. For queueing networks, these are as in the original RCAT, namely that all passive actions be enabled in all states and that all states also have an incoming instance of every active action;

5. The required product-form for state $\underline{s} = (s1\ ,\ s2)$ is now $\pi(\underline{s}) \propto \pi_1(s_1)\pi_2(s_2)$ where $\pi_k(s_k)$ is the equilibrium probability (which may be unnormalised) of state s_k in R_k.

Example [6]

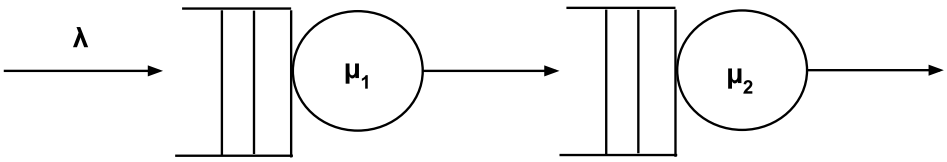

Figure 2.3: A simple tandem queue system

Consider a tandem queue system, as in Figure 2.3, which has 2 M/M/1 queuing nodes where input in queue 2 is coming from output from queue 1. Queue 1 has an external arrival rate of λ and queuing nodes i, where $1 \leq i \leq 2$, have a service rate of μ_i. Let external arrival be represented by action e, internal transfer between queues be action a and departure from the system be action d. This system can be modelled in PEPA as follows:

$$\begin{aligned}
Sys &\stackrel{def}{=} P_0 \underset{a}{\bowtie} Q_0 \\
P_0 &\stackrel{def}{=} (e, \lambda).P_1 \\
P_n &\stackrel{def}{=} (e, \lambda).P_{n+1} + (a, \mu_1).P_{n-1} \\
Q_0 &\stackrel{def}{=} (a, \top).Q_1 \\
Q_n &\stackrel{def}{=} (a, \top).Q_{n+1} + (d, \mu_2).Q_{n-1}
\end{aligned}$$

Using Step 1 of the algorithm and activity substitution we get,

$$\begin{aligned} R_0 &\stackrel{def}{=} (e,\lambda).R_1 \\ R_n &\stackrel{def}{=} (e,\lambda).R_{n+1} + (a,\mu_1).R_{n-1} \\ S_0 &\stackrel{def}{=} (a,x_a).S_1 \\ S_n &\stackrel{def}{=} (a,x_a).Q_{n+1} + (d,\mu_2).Q_{n-1} \end{aligned}$$

Step 2: Now we need to find reversed rates for action type a. Since both the queuing nodes are M/M/1 queues, their equilibrium state probabilities are known to be $\pi_1(q) = (1-\rho_1)\rho_1^q$ for node 1 and $\pi_2(q) = (1-\rho_2)\rho_2^q$ for node 2 (refer equation 2.4), where $\rho_1 = \frac{\lambda}{\mu_1}$ and $\rho_2 = \frac{x_a}{\mu_2}$, since the arrival and service rates are state independent.

$\Longrightarrow$

$$\begin{aligned} \overline{r_a} &= \frac{\pi_1(n+1)r_a^{n+1}}{\pi_1(n)} \\ &= \rho\mu_1 \\ &= \lambda \end{aligned}$$

Step 3: Solving equation $x_a = \overline{r_a}$ and executing step 3 of the algorithm we get,

$$x_a = \lambda$$

Finally, we can calculate the product form solution result by step 5,

$$\begin{aligned} \pi(P_m,\ Q_n) &= \pi(P_m)\pi(Q_n) \\ &= \pi_1(m)\pi_2(n) \\ &= (1-\rho_1)\rho_1^m(1-\rho_2)\rho_2^n \\ &= (1-\rho_1)\rho_1^0(1-\rho_2)\rho_2^0\rho_1^m\rho_2^n \\ &= \pi(P_0,\ Q_0)\rho_1^m\rho_2^n \end{aligned}$$

where where $\rho_1 = \frac{\lambda}{\mu_1}$ and $\rho_2 = \frac{x_a}{\mu_2} = \frac{\lambda}{\mu_2}$. The derived product form solution aggress with Jackson's Theorem (Theorem 2.5.1, equation 2.7) confirming its validity.

2.6.3 Stochastic Petri nets (SPNs)

Stochastic Petri Nets [13] (SPNs) are a popular higher level formalism for Markovian (and other interacting) systems apart from Stochastic Process Algebra like PEPA. They are more expressive in a natural graphical way than SPAs but are more difficult to analyse structurally.

Definition 2.6.7
A stochastic Petri net can be defined as a tuple,
$SPN = (\mathcal{P}, \mathcal{T}, \mathcal{X}(\ .\ ), \mathrm{I}(\ .\ ), \mathrm{O}(\ .\ ), m_0)$ where:

- $\mathcal{P} = \{P_1, \ .\ .\ .\ , P_N\}$ is a set of N places,
- $\mathcal{T} = \{T_1, \ .\ .\ .\ , T_M\}$ is a set of M transitions,
- $\mathcal{X} : \mathcal{T} \to \mathbb{R}^+$ is a positive valued function that associates a firing rate with every transition,
- $\mathrm{I} : \mathcal{T} \to \mathbb{N}^N$ associates an input vector with every transition,
- $\mathrm{O} : \mathcal{T} \to \mathbb{N}^N$ associates an output vector with every transition,
- m is a vector called *marking* which denotes the number of tokens m_i placed in every place P_i. m_0 is the initial marking.

To help analyse SPNs we define a fundamental structure called the *Building Blocks* or BBs, give an expression for its product-form solution and conditions required for its existence.

2.6.3.1 Building blocks

Definition 2.6.8
A SPN S with set of transitions $\mathcal{T}$ and set of N places $\mathcal{P}$ is a **building block** if it satisfies the following conditions:

1. For all $T \in \mathcal{T}$ then either T is an output transition with $O(T) = \emptyset$ or T is an input transition with $I(T) = \emptyset$.
2. For each $T \in T_I$ (set of input transitions), there exists $T' \in T_O$ (set of output transitions) such that $O(T) = I(T')$ and vice versa.
3. Two places $P_i, P_j \in \mathcal{P}$, $1 \leq i, j \leq N$, are connected if there exists a transition $T \in \mathcal{T}$ such that the components i and j of $I(T)$ or of $O(T)$ are non-zero.

Thus condition 1 requires all transitions to be either input or output transitions. Condition 2 required any input transition T_y of the building block feeding a subset of places y to have a corresponding output transition T'_y that consumes the tokens from the same subset y. Finally condition 3 requires the SPN to be connected. Figure 2.4 is an example of a building block with three places $\mathcal{P} = \{P_1, P_2, P_3\}$, three input transitions $T_I = \{T_{12}, T_{23}, T_3\}$ and three output transitions $T_O = \{T'_{12}, T'_{23}, T'_3\}$

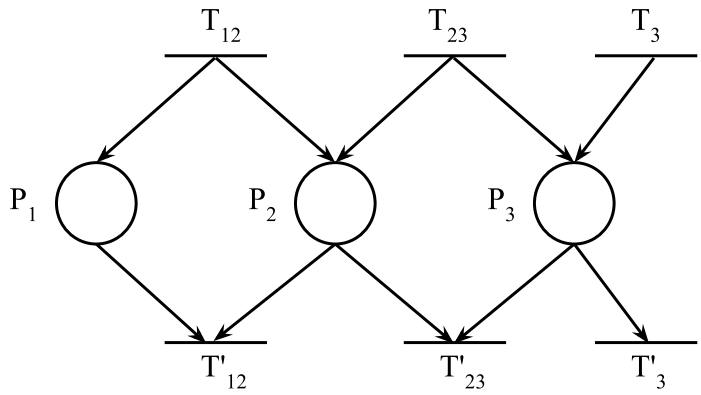

Figure 2.4: Example of a building block

2.6.3.2 Product form of building blocks

A product form result can be derived for an arbitrary building block using ERCAT and is detailed with proof in paper [13]. The paper thus derives a theorem for a product form result of an arbitrary building block, given below.

Theorem 2.6.3
Consider a Building block S with N places and $N \subseteq 2^{1,\ldots,N} \backslash \emptyset$. Let $\rho_y = \frac{\lambda_y}{\mu_y}$ for $T_y, T'_y \in \mathcal{T},\ |y| \geq 1$. If the following system of equations has a unique solution $\rho_i,\ (1 \leq i \leq N)$:

$$\begin{cases} \rho_y = \prod_{i \in y} \rho_i & \forall y\text{: } T_y, T'_y \in \mathcal{T} \ \wedge \ |y| \geq 1 \\ \rho_i = \frac{\lambda_i}{\mu_i} & \forall i : T_i, T'_i \in \mathcal{T},\ 1 \leq i \leq N \end{cases}$$

then the net's balance equations – and hence stationary probabilities when they exist – have product-form solution:

$$\pi(m_1, \ \ldots \ , m_N) \propto \prod_{i=1}^{N} \rho_i^{m_i}$$

.

Please refer to paper [13] for detailed proof of Theorem 2.6.3. Thus the conditions required for the building block in Figure 2.4 in product form are

$$\begin{cases} \rho_{12} = & \rho_1 \rho_2 \\ \rho_{23} = & \rho_2 \rho_3 \\ \rho_3 = & \frac{\lambda_3}{\mu_3} \end{cases}$$

which gives the steady state probabilities and product form unconditionally as as:

$$\pi(m_1, m_2, m_3) \propto \left(\frac{\lambda_{12}\lambda_3\mu_{23}}{\mu_{12}\mu_3\lambda_{23}}\right)^{m_1} \left(\frac{\lambda_{23}\mu_3}{\mu_{23}\lambda_3}\right)^{m_2} \left(\frac{\lambda_3}{\mu_3}\right)^{m_3}$$

From Theorem 2.6.3, the following corollary can be derived.

Corollary 2.6.1
Consider a Building block S in product form as defined in Theorem 2.6.3, let $T_y{}' \in T_O$. The reversed rate of transition $T_y{}'$ is λ_y, which is the the rate of the corresponding input transition T_y.

2.6.3.3 RCAT for Stochastic Petri Nets

We can specify many complex SPNs as a composition of multiple building blocks (BBs). Since the BBs themselves are in product form we can say that:

1. the reversed rates of the reversed actions corresponding to the output transition firings are constant;
2. the input transitions are always enabled;
3. each state of the BB can be reached by the firing of any output transition.

This ensures that the three RCAT conditions hold and we can run Multiple Agent RCAT ([9]) on a composition of several BBs to find a product form for complex SPNs.

2.6.3.4 Practical Example

The Figure 2.5 gives an example of a simple SPN composed of two building blocks. The dotted lines show the passive composition between the two building blocks. The output transitions T'_{12} and T'_{23} from one building block (BB_1) corresponds with input transition T_{45} of the second building block (BB_2). Similarly, output transition T'_5 from BB_2 corresponds with input transition T_{23} from BB_1. The conditions for BB_1 to be in product form are:

$$\begin{cases} \rho_{12} = & \rho_1\rho_2 \\ \rho_{23} = & \rho_2\rho_3 \\ \rho_3 = & \frac{\lambda_3}{\mu_3} \\ \rho_{12} = & \frac{\lambda_{12}}{\mu_{12}} \\ \rho_{23} = & \frac{x_{23}}{\mu_{23}} \end{cases}$$

where x_{23} is the unknown rate for input transition T_{23}. Similarly the conditions for BB_2 to be in product form are:

$$\begin{cases} \rho_{45} = & \rho_4\rho_5 \\ \rho_4 = & \frac{\lambda_4}{\mu_4} \\ \rho_5 = & \frac{\lambda_4}{\mu_4} \\ \rho_{45} = & \frac{x_{45}}{\mu_{45}} \end{cases}$$

where x_{45} is the unknown rate for input transition T_{45}.

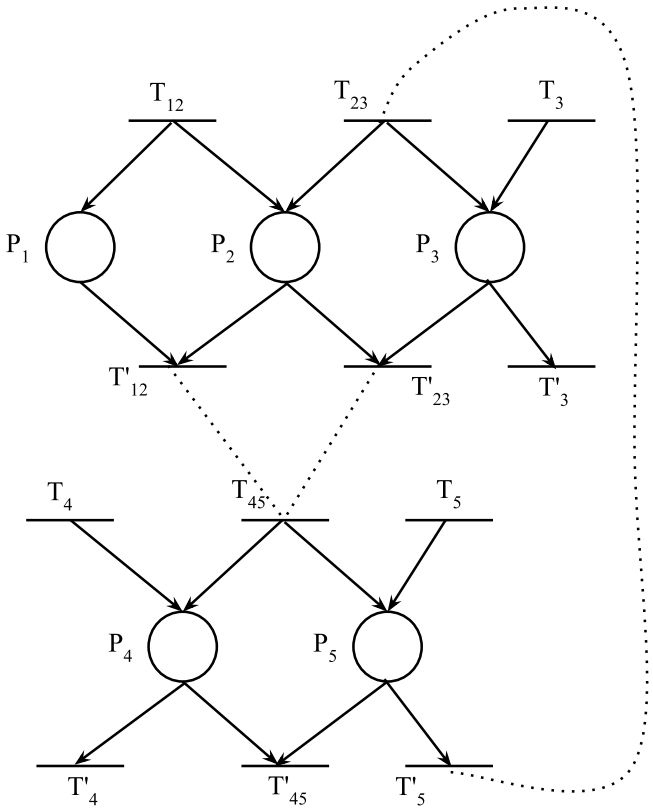

Figure 2.5: A simple SPN with two building blocks

We then derive rate equations for unknowns x_{23} and x_{45} by applying RCAT and using corollary 2.6.1. Thus the rate equations are:

$$\begin{cases} x_{45} = & \bar{\mu_{12}} + \bar{\mu}_{23} = \lambda_{12} + x_{23} \\ x_{23} = & \bar{\mu}_5 = \lambda_5 \end{cases}$$

Substituting the values into the rate equations the conditions for product form are:

$$\begin{cases} (\lambda_{12} + \lambda_5)\mu_5\mu_4 & = \mu_{45}\lambda_4\lambda_5 \\ \lambda_5\mu_2\mu_3 & = \lambda_2\lambda_3\mu_{23} \end{cases}$$

These conditions yield the product form solution:

$$\pi(m_1, m_2, m_3, m_4, m_5) \propto \left(\frac{\lambda_{12}\lambda_3\mu_{23}}{\mu_{12}\mu_3\lambda_5}\right)^{m_1} \left(\frac{\lambda_5\mu_3}{\mu_{23}\lambda_3}\right)^{m_2} \left(\frac{\lambda_3}{\mu_3}\right)^{m_3} \left(\frac{\lambda_4}{\mu_4}\right)^{m_4} \left(\frac{\lambda_5}{\mu_5}\right)^{m_5}$$

Chapter 3

Implementation of the RCAT

This chapter covers the design, implementation and testing aspects of the RCAT implementation. Since RCAT was implemented from scratch, considerable effort was put in to produce easy and scalable application programming interface (API) and clean and modularised code.

3.1 Design Decisions

The first design consideration was the API of the RCAT solver. The inputs to the RCAT algorithm (Theorem 2.6.1) are two PEPA compound agents synchronising over some action labels. Thus while considering the input to the automated version of RCAT, there were two choices:

1. User splits input requiring minimal automated parsing

In this possible implementation, the user is required to write PEPA processes such that they be can directly used in reversed rate calculation with minimal automated parsing. For example, a PEPA description, $P_n = (e, \lambda).P_{n+1}(n \geq 0)$, can be converted to a process structure in MATLAB as shown in Figure 3.1.

```
p( 1 ).definition( 1, : ) =
{ 'n', 'e', 'lambda', 'n+1', 'n>=0' }
```

Figure 3.1: PEPA Process Description converted to MATLAB format

This option is thus easier to program but is non-intuitive and cumbersome to the user. The program would also be prone to calculation errors as we would not be able to robustly validate the input. An additional disadvantage would be coupling the API to close to the functional logic of RCAT. Finally it would also require the user to have some knowledge of the programming language to supply a ready made PEPA structure as input as can be seen from the Figure 3.1.

2. User inputs Pure PEPA descriptions

In this implementation, the user inputs component descriptions as they would to a non-automated RCAT theorem (shown in Figure 3.4). This ensures ease of use and is quite intuitive for the user. It also deals with input validation and decoupling of the API from functional aspects of the RCAT theorem.

Option two was selected for its aforementioned advantages. In further detail, the API is simply a function `RCATscript` which accepts the full PEPA description as text. For example, we write PEPA process description for the RCAT algorithm as

$$\begin{aligned} P_n = &\quad (e, \lambda).P_{n+1} \quad (n \geq 0) \\ P_n = &\quad (a, \mu_1).P_{n-1} \quad (n > 0) \\ Q_n = &\quad (a, \top).Q_{n+1} \quad (n \geq 0) \\ Q_n = &\quad (d, \mu_2).Q_{n-1} \quad (n > 0) \\ &\quad P_0 \underset{a}{\bowtie} Q_0 \end{aligned}$$

This is an example of a queueing network (a basic tandem network with two nodes) modelled in PEPA. Its corresponding translation to code is shown in Figure 3.2. Please note that the unspecified action rate $\top$ effectively has the value of ∞, and is therefore represented as *infinity* in the program.

```
P(n) = (e, lambda).P(n+1) for n >= 0
P(n) = (a, mu1).P(n-1) for n > 0
Q(n) = (a, infinity).Q(n+1) for n >= 0
Q(n) = (d, mu2).Q(n+1) for n > 0
```

Figure 3.2: Pure PEPA Process Description translated to RCAT Program input

From Figure 3.2, it is apparent that with minimal substitution we can translate a pure PEPA process description to code input. RCAT also requires as input the cooperating agents (processes) with the actions they are synchronising on, which are converted into code input as shown in Figure 3.3.

```
P(0) with Q(0) over {a}
```

Figure 3.3: PEPA Cooperating Agents translated to RCAT Program input

$P(0)$ and $Q(0)$ are the cooperating agents and $\{a\}$ is the set of synchronising actions. It is mandatory that the cooperation is written as in Figure 3.3 for parsing.

Running RCAT in MATLAB

The RCAT algorithm is run as shown in Figure 3.4, with the converted PEPA process description (Figure 3.2) in a cell array as the first input and with PEPA cooperation string (see Figure 3.3) as second input.

```
> input1 = { 'P(n) = (e, lambda).P(n+1) for n >= 0',
  'P(n) = (a, mu1).P(n-1) for n > 0',
  'Q(n) = (a, infinity).Q(n+1) for n >= 0',
  'Q(n) = (d, mu2).Q(n+1) for n > 0' }
>
> input2 = 'P(0) with Q(0) over {a}'
>
> RCATscript( input1, input2 )
```

Figure 3.4: Function used to run the RCAT Program

Choice of programming language

The next design consideration was the choice of programming language and effort was made to make a choice comfortable for both the developer and user and meeting the demands of the program. MATLAB was chosen as the implementation language for the project because of its capability to perform symbolic calculations as RCAT operates largely on symbolic variables. Its *Symbolic Math Toolbox* provides a large library of functions for symbolic variable instantiation, substitution, handling and operating on symbolic math expressions. Its greatest advantage is that programs can calculate in terms of symbolic variables giving a symbolic result.

Other languages considered were Python and Java. Python has a symbolic manipulation library called '*sympy*' which is a lightweight normal Python module which aims to be a full-featured computer algebra system. MATLAB was chosen over Python as it is better tested and documented and because of its vast Library of functions. Java despite is object oriented capabilities was not chosen as a symbolic manipulator would have to be written from scratch and robustly tested thus making the task extremely time consuming.

Before starting implementation, we decided to break the RCAT Theorem into smaller implementation tasks. Since the project was not object oriented, we structured the system according to the implementation stages. These undermentioned implementation stages are based on the generic RCAT algorithm detailed in the background (Section 2.6.2).

1. Parsing PEPA input and constructing process structures P_k and R_k
2. Checking that RCAT conditions (1-3) hold for input PEPA model
3. Calculating reversed rates of passive actions
4. Replacing passive actions with symbolic reversed rates
5. Deducing the product-form solution of the model.

3.2 Implementation

3.2.1 Parsing PEPA input

The initial step in implementing the project was parsing the PEPA input and converting it to the process structure P_k. On analysing a process description, we realised that a PEPA process definition can be broken into parts (or process descriptors) such as 'name of the process', 'source state of the transition', ' destination state of the transition', ' process action label', 'action rate', and 'process state domain'. For example, a process definition $P_n = (e, \lambda).P_{n+1} (n \geq 0)$ has P as the name of the process, n as the state P is currently in, $n + 1$ as the the state P is transitioning to, e as its action label, λ as it action rate and $n \geq 0$ as the state domain. We thus parse this information from the process input by using regular expressions.

Thus the program `RCATscript`, on receiving input, a process description string (Figure 3.2), calls the function `registerProcess` with one process description at a time. The program `registerProcess` is responsible for parsing the process string and storing it in a map of processes, ordered by process name. Parsing is done using regular expressions as in Figure 3.5.

```
matches = regexp( processDescription ,
'([A-Z][0-9]*)\((.+)\) = \((.+), (.+)\)\.([A-Z][0-9]*)\
((.+)\)(?: for )?(.*)', 'tokens' );
```

Figure 3.5: Code for parsing PEPA process description using regular expressions

The built in MATLAB `regexp` function allows retrieving matched text from an input string, that corresponds to portions of the regular expression(s) enclosed in parentheses. In further detail, the regular expression `([A-Z][0-9]*)` will match one letter in the upper case and zero or more numbers, thus allowing a process P and a process $P1$ to both be parsed. Regexp `\((.+)\)` will ignore parentheses and match anything within them while `(?: for)?(.*)` will optionally look for the keyword `for` and optionally match anything after it. It thus gives the flexibility of having an optional state domain descriptor for a process.

Running the code in Figure 3.5 on code input- `'P(n) = (e, lambda).P(n+1) for n >= 0'`, we ultimately get a list of aforementioned process descriptors - `{P, n, e, lambda, n+1, n>=0}`.

3.2.1.1 Constructing P_k

P_k is a structure consisting of k PEPA processes with their definitions. $k = 1, 2$ is used in our initial system/network models used as input to RCAT, thus P_k will correspond to two separate PEPA processes analogous to P and Q respectively in Figure 3.2.

Function `addToProcessStructure` stores processes, ordered by process name, in a map called *registeredProcesses*. So multiple descriptions of any process P will be stored under the same key 'P'. The map `registeredProcesses` behaves as the P_k for this implementation of RCAT.

A description of a process with name P comprises of aforementioned process descriptors and is added to `registeredProcesses` as a map with the process descriptors as keys. Figure 3.6 lists the keyset which each process description map is ordered by. If a process P has multiple descriptions (as shown in Figure 3.2), they are converted into maps ordered by process descriptors and stored together in a cell array (a data structure in MATLAB which allows entries of different classes). Thus `registeredProcesses` has a key-value pair : 'process name'-'descriptions cell array'.

```
keyset = { 'transitionFromState','actionName','actionRate',
'transitionToState','domain' };
valueset = { eval(processDefinition{2}), actionLabel,
actionRate, eval(processDefinition{6}),
[domainMin, domainMax] };
```

Figure 3.6: Process description map's keyset and valueset

Values of every process description (as shown in Figure 3.6) have certain properties

1. *transitionFromState* is the source state the transition is coming from while *transitionToStat*e is the destination state for that transition. They are stored as a MATLAB symbolic variables to simplify implementation stages such as RCAT condition checking (Section 3.2.4).
2. *actionName* is stored as a String
3. *actionRate* is stored as MATLAB symbolic variable as it is used extensively in reversed rate calculations (Section 3.2.2).
   (a) RCAT requires all passive action rates (rate = $\top$) to be relabelled to avoid confusion in multiple infinite action rates. We achieve this by relabelling all action rates with *'infinity'* to symbolic variable *'x'* postfixed with the action name of that passive rate. So a process with action rate *'infinity'* and action label *'a'* will be relabelled as *'x_ a'*.
   (b) Action rates are also parsed to check if they are mathematical expressions using function `stringToMatlabExpr`. It parses a string, finds variables in the string, makes them symbolic and then returns the evaluated string as a symbolic variable which is stored as action rate. This requires action rates to compulsorily begin with an alphabet in the lower case and is validated by the same function. While evaluating the action rates, the program makes an assumption that no active action rate can have value 'infinity' as this would

cause the program to assume the action was passive when it was actually active.

4. *domain* is an equality or an inequality mathematical expression. This is analysed to give a range of values for which the state transition holds. The function `parseDomain` achieves this by parsing the (in)equality string and returning a tuple of (*domainMin*, *domainMax*), which denotes the maximum and minimum number of the range that process transition is valid for. We assume the state space for all R_k to be $[0, \infty]$. Thus the maximum domain of a condition string $n > 0$ will be ∞. But the maximum and minimum domain of a condition string $n = 0$ will be evaluated as $[0, 0]$.

The program `addToProcessStructure` also stores all active action names and passive action names for each process in cell arrays and inserts them into maps `activeActionLabels` and `passiveActionLabels` respectively ordered by process name. This simplifies the task of creating the structure R_k.

3.2.1.2 Constructing R_k from P_k

R_k, similar to P_k, is structure consisting of k PEPA processes where $k = 1, 2$. R_k in this implementation is modelled as a MATLAB structure array (an array with named fields that can contain data of varying types and sizes) called `r`. Each entry in the structure `r` contains fields for various properties of the processes which are populated using function `createRk`. The `definitions` field refers to the parsed PEPA descriptions, the `activeLabels` field refers the set of active actions for each P_k and the `passiveLabels` field refers the set of passive actions for each P_k. Figure 3.7 is an example of structure `r` used to model two PEPA processes.

```
r =

1x2 struct array with fields:
    definitions
    activeLabels
    passiveLabels
```

Figure 3.7: Fields in structure r containing r(1) and r(2)

The function `createRk` is used for creating the structure `r`. It uses the maps `registeredProcesses, activeActionLabels` and `passiveActionLabels` that were generated in the function `addToProcessStructure` and the process name (for example P) as the key to instantiate the three fields of the structure `r` structure as shown in Figure 3.8.

```
for i = 1:numOfProcesses
  r(i).definitions
  = registeredProcesses( processKeyset{1,i} );
  r(i).activeLabels
  = setActionLabels(activeActionLabels,processKeyset{1,i});
  r(i).passiveLabels
  = setActionLabels(passiveActionLabels,processKeyset{1,i});
end
```

Figure 3.8: Populating fields of structure R_k

3.2.1.3 Parsing PEPA cooperation

The function `registerCoop` parses input for a PEPA cooperation (synchronisation) between two processes. A cooperation, as shown in Figure 3.3, is the second input to the API function `RCATscript`. The cooperation string is parsed using regular expressions as in Figure 3.9. `registerCoop` returns the action labels the two processes are cooperating over in a cell array called `coopLabels`, which is used in calculating reversed rates and checking RCAT conditions. For input as in Figure 3.3, `coopLabels` will equal $\{a\}$.

```
matches = regexp( coopDescription,
  '([A-Z][0-9]*)\(([^\)]+)\) (with .+\s*)+ over \{(.*)\}',
  'tokens' );
```

Figure 3.9: Code for parsing PEPA Cooperation string using regular expressions

3.2.2 Calculating Reversed Rates

Reversed rates of passive actions are calculated using the formula 2.8 stated in the generic algorithm (Section 2.6.2.3). The formula requires that the steady state probabilities π_k are known for each $k = 1, 2$ in R_k and requires r_a^i, the specified forward rate of action type a going out of state i (for the relevant $k = 1, 2$ in R_k) to be known. Thus the reversed rate calculation is divided into the undermentioned subsections.

3.2.2.1 Calculating steady state probability

The RCAT application was primarily designed to run on systems composed of M/M/1 queues, which have known equilibrium probability distributions and are given by the formula 2.4 stated in Section 2.3.2. The steady state distribution formula uses the utilisation ρ of an M/M/1 queue which is defined as

$$\rho_i = \frac{\lambda}{\mu}$$

where λ is the aggregate arrival rate at node i and μ is the service rate of node i. Since it is assumed for all $k = 1, 2$, R_k is a M/M/1 queue, ρ_k = arrival rate of R_k/ service rate of R_k.

Calculating total arrival and service rates for R_k

The arrival rate (since R_k is M/M/1) is equal to the sum of all rates for transitions from state i to $i+1$ while the service rate is sum of all rates for transitions coming into state i, so from $i+1$ to i. The function used to find arrival and service rates $\forall k$ in R_k is `getAggregateArrivalAndServiceRates`. The calculation involves iterating through all the definitions of a process and determining the the direction of the transition in each definition(see Figure 3.8). Function `isTransitioningForwards` determines if a process transition is going out or coming into state i using process descriptors *transitionFromState* and *transitionToState.* The arrival rate and service rate is the `forwardSum` and `backwardSum` respectively in Figure 3.10.

```
for definition = process.definitions
   if isTransitioningForwards( definition )
      forwardSum = forwardSum + definition('actionRate');
   else
      backwardSum = backwardSum + definition('actionRate');
   end
end
```

Figure 3.10: Code for calculating arrival rate and service rate for each process

Function `sspdMM1` calculates the steady state probability of an M/M/1 queue given an arrival and service rate.

```
syms r x;
rho = ( arrivalRate / serviceRate );
formula = '(1 - r) * r^x';
temp = subs( formula, x, state );
sspd = subs( temp, r, rho );
```

Figure 3.11: Code for SSPD calculation of M/M/1 queue

The formula shown in Figure 3.11 is the formula for the equilibrium probability distribution (see 2.4) of a M/M/1 queue. The MATLAB function `subs` performs a symbolic variable substitution in a given mathematical expression, which in this case is the SSPD formula. On calculating ρ (`rho`) with arrival and service rate (both symbolic variables) and performing symbolic substitution in the formula for steady state probability, we get π_k for each $k = 1, 2$.

3.2.2.2 Calculating specified forward rate

Function `getStatesAndRateForAction` calculates r_a^i, the specified forward rate of action type a going out of state i in the process where a is the active action. Thus the function iterates over process definitions of P as action a belongs to the set of activeLabels in P and returns the rate.

3.2.2.3 Calculating reversed rates

Function `calculateReversedRate` uses the formula 2.8 stated in the generic algorithm (Section 2.6.2.3) to calculate reversed rates for all passive synchronising actions. The code in Figure 3.12 corresponds to this, where the `forwardRate` is the specified forward rate of a given active action, `iStateSSPD` and `jStateSSPD` is the steady state probability at state i and j respectively for some process P. As all three are MATLAB symbolic variables, the function `simplify` reduces the formula which is a mathematical expression of the form $\pi_k(i)r_a^i / \ \pi_k(j)$.

```
formula = (forwardRate * iStateSSPD) / jStateSSPD;
reversedRate = simplify(formula);
```

Figure 3.12: Code for reversed rate calculation

3.2.2.4 Storing reversed rates

As a final step in reversed rate calculation, we need to store the reversed rate for each action a that belongs to the set of cooperating actions, that is $\forall a \in coopLabels$. The function `storeReversedRates` performs the task of storing reversed rates in a map called `reversedRates` with each action in `coopLabels` as the key. The map of reversed rates becomes significant while replacing the passive actions (x_a) with the relevant reversed rates in R_k and checks they are the same at each instance if there are multiple instances of the same action.

3.2.3 Replacing Passive Actions with Reversed Rates

In the structure R_k, all passive actions are represented in the form `x_a` where a is some passive action. These rates have to be replaced by the calculated reversed rates for all actions in the set of cooperating actions, that is $\forall a \in coopLabels$. We substitute symbolic solutions for each rate variable x_a in R_k in the function `setPassiveActionRate` which matches the passive action rate with the right reversed rate and substitutes it in R_k.

```
1 if isequal( definition( 'actionName' ), actionLabel )
2    oldActionRate = definition( 'actionRate' );
3    definition( 'actionRate' ) = reversedRates( actionLabel );
4    newActionRate = definition( 'actionRate' );
5 end
```

Figure 3.13: Code for substituting passive action rates

MATLAB is a pass by value language, thus when a function (modifying a structure field) returns, the caller function's copy of the structure is replaced by the functions copy such that only the modified field is replaced. This also means that MATLAB uses 'copy-on-write', that is, variables are only copied if you modify them. This feature is used in function `setPassiveActionRate` where assigning reversed rate (on Line 3 of Figure 3.13) to variable `definition`, its value changes in the original structure R_k.

3.2.4 Checking RCAT Conditions

The RCAT theorem has three conditions which need to be met for product form solution to exist for any system model. They are stated in Theorem 2.6.1. We have checked all three conditions in this implementation of RCAT. If any of the three conditions are violated, the program will exit with an exception error.

3.2.4.1 Checking First Condition

RCAT first condition states that

First Condition: Every passive action type in $\mathcal{P}_P(L)$ or $\mathcal{P}_Q(L)$ is always enabled in P or Q respectively.

The condition is to ensure that all passive actions a are enabled in every state of the passive process for a. While constructing the structure R_k, we store the passive actions or the set $\mathcal{P}_P(L)$, where P is some process, in the field `passiveLabels` of structure `r`. The function `checkFirstRcatCondition` iterates through all `passiveLabels` for all $R_k, k = 1, 2$ and ensures that all passive actions are enabled. The function `checkActionIsEnabled` checks if action is enabled in all states of a process transition graph for all descriptors of R_k.

It is assumed that the state space of any process is from 0 to ∞. This assumption is made to perform the condition checks on the whole state space of a process, but the assumptions are intuitively reasonable as (mostly for this implementation) the input is networks formed of M/M/1 queues. Despite this assumption, it will be relatively straightforward to extend the function due to decoupling of concerns. Furthermore, even in a closed network the number in a queue is unbounded since it depends on the (given) initial network population

N. N would only ever be needed to calculate the normalising constant which is out of scope of this project.

```
1 if ~isequal( definition( 'domain' ), stateSpace )
2     if allChecksAreOK( definition, allDefs )
3        isEnabled = true;
4     end
5 else
6     isEnabled = true;
7 end
```

Figure 3.14: Code for Checking First Condition

Function `checkActionIsEnabled` (code snippet in Figure 3.14) compares the *domain* of the passive process which has the passive transition with the state space of all processes in R_k. As mentioned before the state space is assumed to be $[0, \infty]$. If the domain equals the state space, then it is trivial that the action is enabled throughout the process transition graph. If not, function `allChecksAreOK` evaluates the domain of the process further. This helper function considers two cases:

1. when there is a passive transition going from $n \rightarrow n + 1$

If a process has a transition $n \rightarrow n+1$ where n is the current state, then for an action to always be enabled, the process description needs a domain of $[0, \infty]$ (that is equal to state space). If not, the first condition will be violated.

2. when there is a passive transition going from $n \rightarrow n - 1$

```
1 if isequal( domain(1), 1 )
2    if isSymbolicEqual( transitionTostate, eval('n-1') )
3       if hasInvisibleTransition( allDefs, definition )
```

Figure 3.15: Code for Checking First Condition

If a process has a transition $n \rightarrow n-1$ where n is the current state, the process description will have a domain of $[1, \infty]$ because of condition string $n > 0$ (checked in Line 1 of Figure 3.15). For an action to be enabled, an additional 'invisible' transition is required going from $n \rightarrow n$ where $n = 0$. Function `hasInvisibleTransition` checks for invisible transitions. If all conditions are satisfied, the function notifies the user that the First condition has been satisfied (as in Figure 3.16).

```
>> RCATscript(x2, y2)
First condition of RCAT is satisfied.

Second condition of RCAT is satisfied.
```

Figure 3.16: Snippet of output of program RCATscript

3.2.4.2 Checking Second Condition

RCAT second condition states that

Second Condition: Every reversed action of an active action type in $\mathcal{A}_P(L)$ or $\mathcal{A}_Q(L)$ is always enabled in $\overline{P}$ or $\overline{Q}$ respectively.

This condition checks if there is an incoming active action a in every state of the active process for a, $\forall a \in coopLabels$. The rationale is very similar to the first condition. The function `checkSecondRcatCondition` iterates through all `activeLabels` for all $R_k, k = 1, 2$ and using `checkForIncomingTransitions` function checks to see if there is an incoming active action a in every state of the active process for a. On further analysing, it is apparent that *incoming transitions* refer to transitions going from state $n \rightarrow n-1$. This assures that every state n will have an incoming transition for the state space which is assumed to be $[0, \infty]$ for all processes in R_k (since we are primarily dealing with MM1 queues). The *domain* for these transitions is calculated as $[1, Inf]$ and is checked in the function as shown in Figure 3.17 (line 1).

```
if isequal( domain(1), 1 ) && isequal( domain(2), Inf )
   if isSymbolicEqual( transitionTostate, eval('n-1') ) &&
   isSymbolicEqual( transitionFromState, eval('n') )
       isEnabled = true;
   end
```

Figure 3.17: Code for checking incoming transitions

The function `isSymbolicEqual`(Figure 3.18) checks if the states *from* and *to* are equal to n and $n-1$. Finally if all conditions are satisfied, the function notifies the user that the second condition has been satisfied (as in Figure 3.16).

3.2.4.3 Checking Third Condition

RCAT third condition states that

Third Condition: Every occurrence of a reversed action of an active action type in $\mathcal{A}_P(L)$ or $\mathcal{A}_Q(L)$ has the same rate in $\overline{P}$ or $\overline{Q}$ respectively.

This condition checks that the total reversed rate of all incoming active actions a in state k of the active process is equal to a constant $\bar{r_a}$ independent of state k. It is known that for M/M/1 queues, the equilibrium state probabilities are state independent. Thus the reversed rate is same for all the states the transition corresponds to as the ratio of $\pi(n+1)/\pi(n)$ (used in rate equation 2.8) is constant.

We then need to check if there are multiple transitions with the same action name in the description of the active process, calculate their reversed rates and ensure they are equal. Thus if active process P has multiple transitions for action type a, their reversed rates will be calculated and checked to see if they are equal. The function `checkThirdRcatCondition` iterates over all *activeLabels* for all active processes and checks if action a has multiple transitions, $\forall a \in coopLabels \ \& \ a \in activeLabels$. Then its reversed rates are calculated and are checked for equality with function `isSymbolicEqual`. The function checks to see if the difference (Line 2, Figure 3.18) between two symbolic variables is zero, proving they are equal.

```
1 function ret = isSymbolicEqual( s1, s2 )
2     ret = ( simplify( s1 - s2 ) == 0 );
3 end
```

Figure 3.18: Code to checking symbolic equivalence

Finally if all requirements are satisfied, the function notifies the user of satisfying the third condition and concludes RCAT condition checking.

3.2.5 Generating a Product Form Result

All R_k $(k \geq 2)$ agents to RCAT are M/M/1 queues with $\rho_k = \frac{\text{arrival rate of } R_k}{\text{service rate of } R_k}$. Hence the unnormalised equilibrium probability for state i_k in the process R_k is $(\rho_k)^{i_k}$. From this we can derive the product form result using the Jackson's theorem :

$$\pi(m_1, \ . \ . \ . \ , m_N) \propto \prod_{i=1}^{N} \rho_k^{i_k}$$

The RCAT implementation prints a system of rate equations as output by retrieving them from structure R_k as a part of the implementation stage 'Replacing Passive Actions with Reversed Rates'. This system of rate equations is solved and their values are replaced in ρ_k to generate the aforementioned product form result. Thus for instance if we derive the rate equation - '$x_a = \lambda$', we replace the value of unknown action rate x_a of action a with λ and use the new value while generating the product form result. We can use MATLAB functions `solve` to evaluate solutions to a system of rate equations.

For example, the equations in Figure 3.19 need to be evaluated as they have unknowns in their solutions (lines 2-3). The MATLAB library function `solve` finds a solution for both x_a1 and x_a2 as shown in lines 9 - 15. Before using the function `solve`, all variables must be declared symbolic using function `syms` as shown in Figure 3.19 on line 7. Thus evaluating rate equations, displayed by the software, is left to the user as it is a straightforward process.

```
1  > RCATscript(x2, y2)
2    Printing passive action rates...
3    Reversed rate for passive action a1:
4    x_a1 = p21*(lambda2 + x_a2)
5    Reversed rate for passive action a2:
6    x_a2 = p12*(lambda1 + x_a1)
7
8  > syms x_a1 x_a2 p21 lambda2 p12 lambda1
9
10 > solve('x_a1 = p21*(lambda2 + x_a2)',
11         'x_a2 = p12*(lambda1 + x_a1)' )
12   ans = x_a1: [1x1 sym]
13         x_a2: [1x1 sym]
14
15 > x_a1 = -(p21*(lambda2 + lambda1*p12))/(p12*p21 - 1)
16
17 > x_a2 = -(lambda1*p12 + lambda2*p12*p21)/(p12*p21 - 1)
```

Figure 3.19: Solving system of rate equations

In some cases, the rate equations are non-linear. To solve non-linear system of equations, we can use MATLAB Library function `fsolve`. It checks if the equations converge to a single value and has the option of running multiple iterations with guessed values. Another alternative is to use 'fixed point iteration', but this has not been implemented as a part of this project.

3.3 Testing and Verification

Testing and verification of program logic and output are important aspects of the implementation process as they help in ensuring a robust and correct program. The program logic was unit tested using MATLAB unit test library *xUnit*. The implementation of different parts of the program logic is in different files making it quite easy to be unit tested. All tests follow a similar format where we compare the results of the function being tested with the expected values. *xUnit* provides functions to test exceptions and equality.

```
1 assertEqual( reversedRates.length(), 1);
2
3 assertTrue( isequal( reversedRates( 'a' ), lambda ) );
4
5 assertExceptionThrown(@()storeReversedRates(coopLabels,r),
6 'RCATscript:InvalidComputationStoreReversedRates' );
```

Figure 3.20: Code snippet for a Test

Figure 3.20 is a code snippet from a test `'testStoreReversedRates'` which tests if reversed rates are calculated and stored correctly. *xUnit* provides a function `assertExceptionThrown` to test if the program throws the right exception on being given erroneous input (as shown on lines 5-6 in Figure 3.20). This function is very useful for testing program validations. Please refer to Appendix for additional test cases and instructions to run the unit tests.

Program output is verified by running RCAT on various system models, taken from research papers and provided by the supervisor, and comparing the solutions against the given solutions. The implementation has a command line interface which has been tested by the user. User testing by primarily myself and my supervisor has helped correct any erroneous or unexpected behaviour by the program.

This concludes the implementation of RCAT. The next chapter details the implementation of running RCAT on multiple compound agents and Stochastic Petri Nets.

Chapter 4

Implementation of MARCAT and RCAT for SPNs and PITs

This chapter covers the implementation of MARCAT, an extension of RCAT and the design and implementation of RCAT for Stochastic Petri Nets (SPNs). It also delineates Propagation of Instantaneous Transitions (PITs) and provides an implementation design for running RCAT on PITs.

4.1 Multiple Agents RCAT

The Reversed Compound Agent Theorem (RCAT) is used to derive the reversed process of a cooperation between two agents. RCAT can be generalised to a cooperation of multiple agents. This generalised theorem is referred to as the MARCAT or Multiple Agents RCAT. The theorem is detailed in paper [9].

To implement MARCAT, we extended the current implementation of RCAT. Due to the scalable design of the software, extending the implementation was a straightforward process.

4.1.1 Parsing User Input

The API for this implementation has been maintained as the `RCATscript`, a function with two inputs - a list of PEPA process descriptions and the names cooperating processes. This is detailed in section 3.2.1 of this report. Since the first input is a list of process descriptions, this list has been simply extended to include $k > 2$ number of agents. No changes have been made to the syntax of specifying an agent as input. Processes are parsed to generate the structure R_k in the function `registerProcess`. Since the function parses the process string and stores it in a map of processes ordered by process name, no extensions were required to ensure it worked for multiple processes. As shown in Figure 4.1, we iterate over the processes P_k given as input and register them using the `registerProcess` function.

```
for i = 1:length( processList )
process = processList(i);
registerProcess( registeredProcesses, activeActionLabels,...
passiveActionLabels, process{1} );
```

Figure 4.1: Code to register multiple agents

Creating R_k

R_k, as in the aforementioned implementation of RCAT, is a structure consisting of k PEPA processes where $k \geq 2$. It is modelled as a MATLAB *struct* with fields as mentioned in section 3.2.1.2. To accommodate multiple agents the function createRk was extended to iterate over all the processes in the map of registered processes as opposed to having a fixed size of 2. This is shown in Figure 4.2.

```
for i = 1:numOfProcesses
  r(i).definitions = registeredProcesses(processKeyset{1,i})
```

Figure 4.2: Code for creating R_k

Parsing PEPA cooperation

Since MARCAT is the cooperation of multiple agents, the second input to `RCATscript` has been changed to include the multiple agents. This does not affect the aforementioned method of running the RCAT solver or specifying cooperation between two agents. We specify cooperation $\underset{L}{\bowtie} P_k$, $k \geq 2$ as shown in Figure 4.3. It specifies a cooperation string between three processes P1, P2 and P3. Keyword 'with' is used as the cooperation symbol. The actions which processes cooperate over are stated in parentheses.

```
'P1(0) with P2(0) with P3(0) over {a1, a2, a3}'
```

Figure 4.3: Cooperation String for Three agents

To parse the cooperation string, function `registerCoop` had to be extended. Parsing the cooperation string is achieved using regular expressions as shown in Figure 4.4. String '([A-Z][0-9]*)\ (([^\)]+)\)' matches a single occurrence of process P1(0). Regular expression '(with .+\s*)+' will match multiple occurrences of a string beginning with the keyword 'with'. This is then parsed to divide the string into the names of the processes and the list of cooperating action labels in the same way as mentioned in section 3.2.1.3.

```
matches = regexp( coopDescription, ...
'([A-Z][0-9]*)\(([^\)]+)\) (with .+\s*)+ over \{(.*)\}',
'tokens' );
```

Figure 4.4: Code for Parsing Cooperation String

The implementation for calculating reversed rates and checking RCAT conditions is the same as mentioned in the previous chapter. We can use the rate equations generated to construct the product form result using the same method as detailed in section 3.2.5.

4.2 Generating Product-Form Solutions for Stochastic Petri Nets

Stochastic Petri Nets (SPNs), as mentioned in the section 2.6.3, are different from Stochastic Process Algebra like PEPA and thus cannot be directly input into the RCAT solver. We generate product-form solutions for SPNs by applying MARCAT on a model consisting of several SPN building blocks. For a practical example, please refer to the background.

4.2.1 Formalising Stochastic Petri Nets

Our first concern while implementing this extension was designing a formalism for SPN building blocks. Due to the graph like structure of SPNs, it is difficult to formalise them using a Markovian Process Algebra like PEPA. So a new formalism was designed which treats a subset of SPNs as a connection of building blocks.

4.2.1.1 Formalising building blocks

A building block as defined in *Definition 2.6.8* is an SPN S with set of transitions $\mathcal{T}$ and set of N places $\mathcal{P}$ where $\mathcal{T}$ can be broken into set T_I of input transitions and set T_O of output transitions. So we defined a building block as a MATLAB *struct* as shown in Figure 4.5 with three properties:

1. *places*: This is a set of N places of a building block stored as a cell array of strings.
2. *inputs:* This is the set T_I of input transitions, that is stored as a map with the transition name as the key and the transition firing rate as the value.
3. *outputs:* This is the set T_O of output transitions, that is stored as a map with the transition name as the key and the transition firing rate as the value.

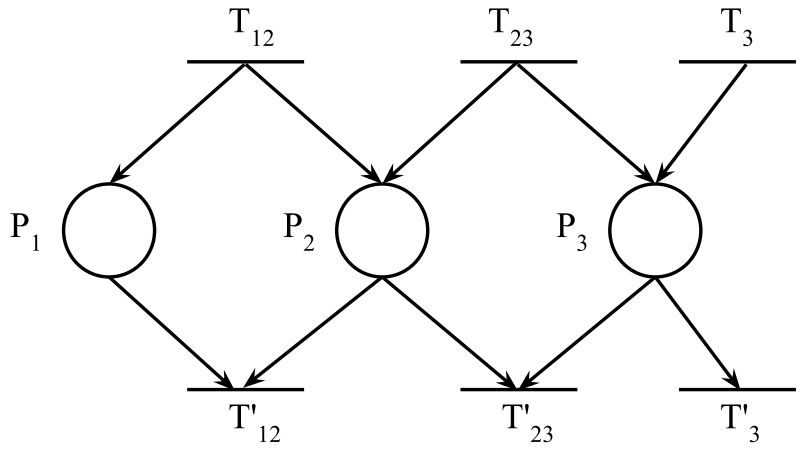

Figure 4.6: Example of a building block

```
1x1 struct array with fields:
  places: {'P1'  'P2'  'P3'}
  inputs: [3x1 containers.Map]
  outputs: [3x1 containers.Map]
```

Figure 4.5: Building Block formalism as a MATLAB struct

4.2.1.2 Specifying Building blocks as input

On determining a way for formalising the building block, we then dealt with converting a structural building block into a programmable input and parsing it to construct the building block *struct*. This is best explained using an example. Let us take the simple building block of Figure 4.6.

It has three places $\{P_1, P_2, P_3\}$ with input transitions $\{T_{12}, T_{23}, T_3\}$ and output transitions $\{T'_{12}, T'_{23}, T'_3\}$. Let the rates of the three input transitions be $\{\lambda_{12}, \infty, \lambda_3\}$ and the rates of output transitions be $\{\mu_{12}, \mu_{23}, \mu_3\}$. Rate ∞ denotes that the firing of an output transition of some building block corresponds to the firing of the transition with rate ∞. The user specifies this information as input to the program `RCATscriptForSPN` as the string shown in Figure 4.7.

```
'{P1, P2, P3}, {i_t12, i_t23, i_t3},
 {lambda12, infinity, lambda3},
 {o_t12, o_t23, o_t3}, {mu12, mu23, mu3}'
```

Figure 4.7: Building Block as input by user

It is important to note that all input transitions are prefixed by 'i' while all output transitions are prefixed by 'o'. This helps the program ensure that

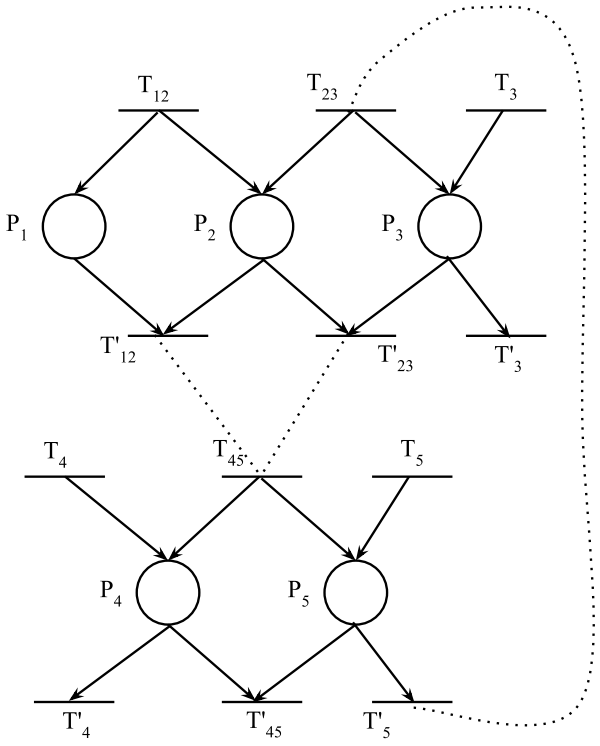

Figure 4.8: Example of a SPN with two building blocks

every input transition has a corresponding output transition, as it is one of the mandatory conditions for an SPN to be a building block. The string is always arranged in the order as shown in the Figure 4.7, starting with places followed by input transitions, input transition rates, output transitions and output transition rates.

4.2.1.3 Specifying Building Block connections

Since an SPN is specified as a model composed with several connected building blocks, the user inputs the several building blocks as a list to the program `RCATscriptForSPN`. The user also has to input a routing probability matrix giving the connections between the different building blocks and their routing probabilities. The Figure 4.8 gives an example of a simple SPN composed of two building blocks. The dotted lines show the passive composition between the two building blocks. So the firing of transitions T'_{12} and T'_{23} from one building block (BB1) corresponds with input transition T_{45} of the other building block (BB2). Similarly, output transition T'_5 from BB2 corresponds with input transition T_{23} of BB1.

The passive cooperations between the building blocks are specified as shown in Figure 4.9.

```
'{o_t12 = 1, o_t23 = 1 to i_t45} ; {o_t5 = 1 to i_t23}'
```

Figure 4.9: User input specifying passive compositions between BBs

Each connection is stated by the user in parentheses {...} separated by a semi-colon as shown in Figure 4.9. Connection '`o_t12 = 1, o_t23 = 1 to i_t45`' states that output transitions $t12$ and $t23$ correspond to input transition $t45$ with routing probability equal to 1. This routing probability is used while calculating rate equations. The routing probability becomes significant while looking at the behaviour of the BB. It the routing probability were p then the output transition has a chance of synchronising with its corresponding input transition with probability p.

4.2.1.4 Running RCAT for SPNs

```
1  listOfBBs = {
2  '{P1, P2, P3}, {i_t12, i_t23, i_t3},
3   {lambda12, infinity, lambda3},
4   {o_t12, o_t23, o_t3}, {mu12, mu23, mu3}',
5  '{P4, P5}, {i_t4, i_t45, i_t5},
6   {lambda4, infinity, lambda5},
7   {o_t4, o_t45, o_t5}, {mu4, mu45, mu5}'}
8
9  connectionString =
10 '{o_t12 = 1, o_t23 = 1 to i_t45} ; {o_t5 = 1 to i_t23}'
11
12 RCATscriptForSPN( listOfBBs, connectionString )
```

Figure 4.10: Running RCAT on a SPN with two building blocks

Figure 4.10 shows how the input is specified for a SPN model (of Figure 4.8) using lists of building blocks and their connections (lines 1-10 in Figure 4.10). This is then run using the API function `RCATscriptForSPN` which takes the list of building blocks and their connections as the two arguments as shown in line 12 of Figure 4.10. It might be worth noting that the M/M/1 queue is simply a BB with one place. Thus the application of RCAT to SPNs is a true generalisation.

4.2.2 Parsing user input

4.2.2.1 Parsing Building Block String

We started with writing a parser for converting the user input of the list of building blocks into a MATLAB *struct* to match the formalism outlined in section 4.2.1.1. This is achieved in the function `createBbStruct` which takes

the list of building block strings as input and returns a *struct* (as shown in Figure 4.5) as output. We use regular expressions for the majority of the parsing in the function `parseBB`.

```
1  match = regexp( bbString, ...
2      '{(.+)}, {(.+)}, {(.+)}, {(.+)}, {(.+)}', 'tokens' );
3  matches = match{1};
4
5  places = regexp( matches{1}, '\s*,\s*', 'split' );
6
7  inputKeyset = regexp( matches{2}, '\s*,\s*', 'split' );
8  rateStrings = regexp( matches{3}, '\s*,\s*', 'split' );
9
10 i_rates = relabelPassiveRates( inputKeyset, rateStrings );
11 inputValues = convertToSymbolicRates( i_rates );
12
13 inputs = containers.Map( inputKeyset, inputValues );
```

Figure 4.11: Parsing building block String

Figure 4.11 gives a code snippet for parsing a building block string. Lines 1-2 take a building block string of the form '{P1, P2, P3}, {i_t12, i_t23, i_t3}, {lambda12, infinity, lambda3}, {o_t12, o_t23, o_t3}, {mu12, mu23, mu3}' and divide it into five different sections; from these sections we derive information about the places and transitions of the building block. MATLAB function `regexp` on lines 1-2 is used to match any text in parentheses using regular expression '{(.+)}'. It is important to note that the building block must be specified as stated in section 4.2.1.3 else the parsing will fail.

On dividing the string into different sections, we then go on to extract values for the set of building block places in line 5. Here the MATLAB function `regexp` splits string *'P1, P2, P3'* into three substrings 'P1', 'P2', 'P3'. Lines 7-8 perform the same type of split on the set of input transition names and input transition rates.

Transition rates which are passive are relabelled to a symbolic variable *'x'* postfixed with the transition name of that passive rate. This is performed in the function `relabelPassiveRates` as shown on line 10 in Figure 4.11. Input transition rates need to be converted into MATLAB symbolic expressions which is achieved by the function `convertToSymbolicRates`, which used the function `stringToMatlabExpr` (a helper function used for converting strings to MATLAB symbolic expressions).

We finally generate *inputs,* a field of the building block *struct*, as a map of the transition names as keys and their rates as values (as shown on line 14 of Figure 4.11). The strings of output transitions and rates are parsed similarly to create *outputs*, the last field of the building block *struct* shown in Figure 4.5.

4.2.2.2 Parsing Building Block connections

On constructing a building block *struct*, the next step is to parse the building block connection string. This is performed by function `parseConnections` which takes a connections string as input and returns a map of connections as output. The map of connections has the destination as the key and the list of inputs as values. For instance, for connection *'o_t5 = 1 to i_t23'*, the key will be i_t23 and the value will be the tuple (o_t5,1). Figure 4.12 is a code snippet of the `parseConnections` function.

```
1 match = regexp( connectionString, '\s*;\s*', 'split' );
2
3 for i = 1:length( match )
4   matches = regexp( match{i}, '{(.+) to (.+)}', 'tokens' );
5   ...
```

Figure 4.12: Parsing building block connections

Parsing is done using regular expression; line 1 in Figure 4.12 shows how the string of connections is split into individual sub-strings of connections each of which are then parsed using a loop. The regular expression '{(.+) *to* (.+)}' retrieves text from either side of 'to'. Thus we separate the key and value pairs for our map of connections.

In the function `parseInputs`, each input value of type '*o_t5 = 1*' is parsed and is separated into transition name string 'o_t5' and MATLAB symbolic expression '1'. The function returns a list of such input transitions names and probability tuples, which is then allocated as the value to its corresponding key in the connection map. Again since we are using regular expressions, the connections string has to be specified as mentioned in section 4.2.1 else the program will terminate because of a parsing error.

4.2.3 Generating Rate Equations

To decide if a composition of building blocks has a product-form, we need to find the unknown (i.e. passive) rates to the input transitions. This is done by setting each unknown rate is to the reversed rate of the corresponding output transition. The system of equations thus generated are the rate equations desired.

This calculation is performed in the function `calculatePassiveRatesForBBs` where for each unknown rate, we derive a rate equation by using the function `getRateEquations`. Figure 4.13 is a code snippet of `getRateEquations`.

```
1 for i = 1:length( connectorList )
2   map = connectorList{i};
3   label = map.keys;
4   passiveActionRate =  passiveActionRate + ...
5   getReversedRateForBB(bbStruct, label{1}) * map(label{1});
6 ...
```

Figure 4.13: Generating rate equations

To find the passive rate of the input transition we use the connections map generated while parsing. The connections map has the name of the input transition with unknown rate as the key and a list of 'connectors' as the value. This list is input to the `getRateEquations` function. Each connection is stored as a map with the corresponding output transition name as the key and its routing probability as the value. We retrieve this key in line 3 of Figure 4.13. and find the reversed rate of the corresponding output transition using function `getReversedRateForBB`. We then multiply this with the routing probability - `map( label{1} )` - as shown in line 5 of Figure 4.13.

A code snippet of function `getReversedRateForBB` is shown in Figure 4.14. To find the reversed rate of a transition we use the building block corollary 2.6.2 stated in section 2.6.3.2 - 'The reversed rate of transition T_y' is λ_y, i.e. the rate of the corresponding input transition'. The corresponding input transition is identified by using regular expressions and prefixing the transition name with 'i' as described in line 1 and 3 of Figure 4.14. Thus an output transition of the form 'o_t23' gives an input transition label 'i_t23'. The rate of this input transition is calculated by iterating over the building block *struct* in function `getActionRateForSPNs`.

```
1 matches = regexp( actionLabel, '\s*_\s*', 'split' );
2 assert( strcmp( matches(1), 'o' ) )
3 inputActionLabel = strcat( 'i', '_', matches{2} );
4 revRate = getActionRateForSPNs(inputActionLabel , bbStruct);
```

Figure 4.14: Finding reversed rate of corresponding output transition

As a final step in generating rate equations, we print the generated equations on standard output which can be solved easily by the user using standard MATLAB tools.

4.2.4 Checking Building Block conditions

For an arbitrary building block to have a product form, conditions detailed in Theorem 2.6.3 need to be satisfied. These conditions are generated and checked in function `checkBBconditions`. The process of validating the conditions is divided into two parts:

1. If the number of input and output transitions (i.e. $size(T_I)$ and $size(T_O)$) is less than or equal to the number of places in the building block, we know the conditions are unconditionally satisfied. This follows directly from the nature of the system of equations generated by the conditions detailed in Theorem 2.6.3. Code for this check is shown in Figure 4.15. We loop over all the building blocks the SPN is composed of and also ensure their input and output transitions are equal as shown in line 1-3.

2. If the number of input and output transitions exceeds the number of places in the building block, then we generate the additional conditions required from Theorem 2.6.3. The Function `furtherConditionCheck` generates and displays the conditions to be satisfied. It is important to note that these conditions need to be solved along with the rate equations and if they are compatible then a product form solution will exist.

```
1 for i = 1:length( bbStruct )
2   assert( length(bbStruct(i).outputs) ==
3           length(bbStruct(i).inputs) );
4   if ~( length(bbStruct(i).places) >=
5         length(bbStruct(i).inputs) )
6     furtherConditionCheck( bbStruct(i) )
7 ...
```

Figure 4.15: Check Building Block Conditions

4.2.5 Generating Product Form Solution

A system of rate equations has been derived for a given SPN using the method in section 4.2.3, where each unknown rate is set to the reversed rate of the corresponding output transition. Theorem 2.6.3 gives conditions for a BB to have a product form which are derived by the aforementioned method in section 4.2.4. To find a product form for the given SPN, the system of equations is solved and if a solution exists then it is checked for compatibility with the product-form conditions of each BB. If compatible, the SPN has a product-form.

As mentioned in section 3.2.5, MATLAB functions `solve` and `simplify` can be used to evaluate solutions to the system of rate equations and conditions generated. The process of evaluating the rate equations remains the same as section 3.2.5. On finding a solution, the product form results of all the participating BBs is combined to give one result. So for a SPN with N places the product form result will be:

$$\pi(m_1, \ldots, m_N) \propto \prod_{i=1}^{N} \rho_i^{m_i}$$

where ρ_i is recalculated substituting the unknowns with solutions to the system of rate equations.

4.3 Chains of interactions between queues

This section provides a background on finding product form solutions for chains of interactions between queues. 'Product-forms in multi-way synchronisations' [14] is a paper which provides the detailed theory about the concepts and algorithm discussed here. Due to time constrains, an implementation of generating product forms for chains of interactions could not be achieved, but we include a high level analysis of how it can be done by extending the current implementation of RCAT.

4.3.1 Background

The paper [14] talks about chains of interactions between queues; for example, negative customers can move a customer from a non-empty queue to another queue chosen probabilistically thus triggering two (or more) queues (or constituent processes) to change their states simultaneously, causing *chains* of instantaneous state changes. The paper explains how such synchronisations can be modelled as the Propagation of Instantaneous Transitions (PITs) [14] to specify the composed models as successive pairwise synchronisations and thus derive product forms by an iterative application of RCAT.

4.3.1.1 PEPA formalism and generating rate equations

The paper [14] introduces a new PEPA construction for representing PITs as follows:

$$P = (a \rightarrow b, \top).Q \tag{4.1}$$

The equation denotes a passive action with type a that takes process P to Q and instantaneously synchronises to active on type b.

Deriving rate equations for the instantaneously synchronising types is difficult and thus an algorithm has been established in the paper [14] to perform rate equation generation with ease. The Algorithm is recursive in nature and considers each synchronising type a, removes its synchronisation from the model by replacing it by type τ in the passive component with the corresponding reversed rate. It simultaneously generates the rate equations corresponding to x_a. It returns the set of rate equations for all cooperating passive actions on termination.

4.3.2 Design and analysis for implementation

In our approach to implement RCAT for PITs, we divided the task into two parts - Parsing the PEPA descriptions and deriving rate equations. While we have not been able to implement this extension, the ideas for designing the implementation have been documented in the following sections. Due to the complexity, the design view will be clearer by using an example [14]. Let us take a G-network with a negative customer trigger as shown in Figure 4.16.

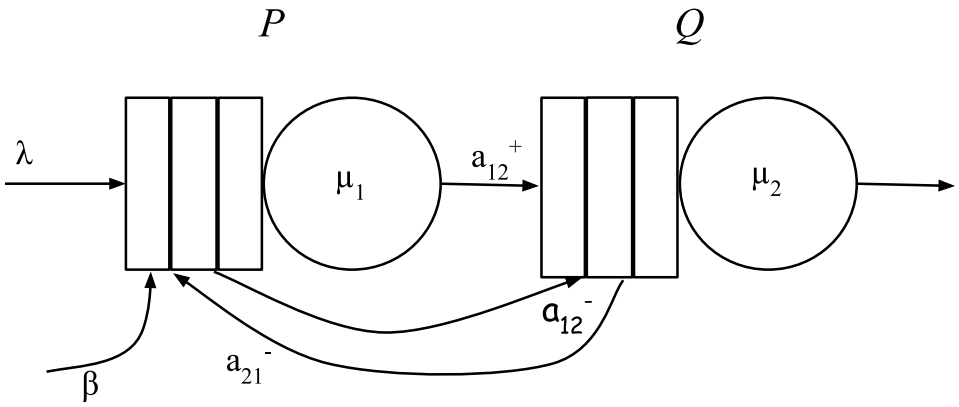

Figure 4.16: G-network with a negative customer trigger

In this network, we have queues P and Q having exponential service times with parameters μ_1 and μ_2 respectively and external positive arrivals to process P with rate λ. PITs are started in P with a negative customer trigger with rate β which propagates to Q and so on until either queue is empty on arrival of the PIT.

4.3.2.1 Parsing user input

The G-network from Figure 4.16 is represented in PEPA using the formalism 4.1 for representing PITs as follows:

$$
\begin{aligned}
P_n &= (\tau, \lambda).P_{n+1} && (n \geq 0)\\
P_n &= (a_{12}^{+}, \mu_1).P_{n-1} && (n > 0)\\
P_n &= (a_{12}^{-}, \beta).P_{n-1} && (n > 0)\\
P_n &= (a_{21}^{-} \rightarrow a_{12}^{-}, \top).P_{n-1} && (n > 0)\\
P_0 &= (a_{21}^{-}, \top).P_0
\end{aligned}
$$

$$
\begin{aligned}
Q_n &= (a_{12}^{+}, \top).Q_{n+1} && (n \geq 0)\\
Q_n &= (\tau, \mu_2).Q_{n-1} && (n > 0)\\
Q_n &= (a_{12}^{-} \rightarrow a_{21}^{-}, \top).Q_{n-1} && (n > 0)\\
Q_0 &= (a_{12}^{-}, \top).Q_0
\end{aligned}
$$

$$P_0 \underset{a_{12}^{+}, a_{12}^{-}, a_{21}^{-}}{\bowtie} Q_0$$

From the PEPA description we can see that the only addition to the current PEPA in the MATLAB parser would be parsing expressions of the type 4.1. We can extend the current parser to store action types $a_{21}^{-} \rightarrow a_{12}^{-}$ with their rates as symbolic variables similar to any other action type. As the algorithm requires the PITs to be distinguished from normal action types, we aim to create a map

of all action types $a \rightarrow b$ where a is the key and b is the value. We add an extra field to our process structure R_k to store these actions separately making iterating over them easy.

Actions of type 4.1 always have a passive action rate specified as input as 'infinity'. Our current PEPA parser performs relabelling of such rates to avoid confusion while deriving rate equations. To relabel rates of PITs we use the same implementation but instead of relabelling the rate of action type $a \rightarrow b$ to $x_{a \rightarrow b}$ we relabel it as x_a. This can easily be done by using regular expressions on action types $a \rightarrow b$.

4.3.2.2 Deriving rate equations

On parsing the PEPA input and generating process structure R_k as a MATLAB *struct*, we implement the algorithm given in the paper [14] to derive rate equations. The algorithm has an empty set of rate equations initially. It starts by iteration over cooplabels, i.e. synchronising actions, and derives rate equations for each using the rate equation 2.8. The algorithm needs to be implemented as a separate MATLAB function and needs to be used instead of the current reversed rate calculator to derive rate equations. The complexity of the algorithm lies in the activity substitution which it performs at each step for all action types. We can use the map of PIT action types generated in the parsing stage to aid with the activity substitution.

This concludes the design and implementation of the project. In the next chapter, we evaluate the usefulness and correctness of the implemented RCAT, MARCAT and RCAT for Stochastic Petri Nets.

Chapter 5

Evaluation

In this chapter, the implementation of RCAT is tested over several queuing networks and Stochastic Petri Nets and their results are verified using research papers and by hand.

5.1 Tandem Queues

An example of a simple queuing network is a network with two queues (nodes) in tandem (Figure 5.1) with Poisson arrivals to queue one at the rate λ. Customers proceed immediately to queue 2 and depart the system on leaving queue 2. Both queues have exponential service times with parameters μ_1 and μ_2 respectively.

This network can be described in PEPA as follows and in as MATLAB code in 3.2.

$$\begin{aligned} P_n &= (e, \lambda).P_{n+1} \quad (n \geq 0) \\ P_n &= (a, \mu_1).P_{n-1} \quad (n > 0) \\ Q_n &= (a, \top).Q_{n+1} \quad (n \geq 0) \\ Q_n &= (d, \mu_2).Q_{n-1} \quad (n > 0) \\ & P_0 \underset{a}{\bowtie} Q_0 \end{aligned}$$

Running `RCATscript` on this queuing network we get reversed rates as shown in Figure 5.2. From the cooperation, it is clear that the three RCAT conditions

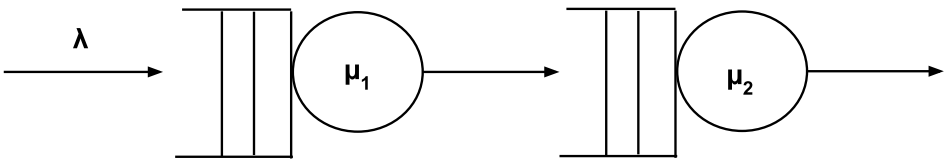

Figure 5.1: Tandem Queue with two Queues

are satisfied on this network. The program checks this and displays a message for satisfying all conditions.

```
1 Reversed rate for passive action a: x_a = lambda
```

Figure 5.2: Passive rates in R_k in Two node Tandem Queuing network

Paper [3] confirms the correctness of the solution generated.

5.2 Feedback Queues

We now consider an example of a generally connected pair of queues with feedback from the paper [3]. The network has two queues, with λ_1 and λ_2 as respective external arrivals and μ_1 and μ_2 as respective service rates. The network has routing probability p_{12} from queue 1 to queue 2 and p_{21} from queue 2 to queue 1. Customers leave the network with probabilities $1-p_{12}$ and $1-p_{21}$ as shown in Figure 5.3.

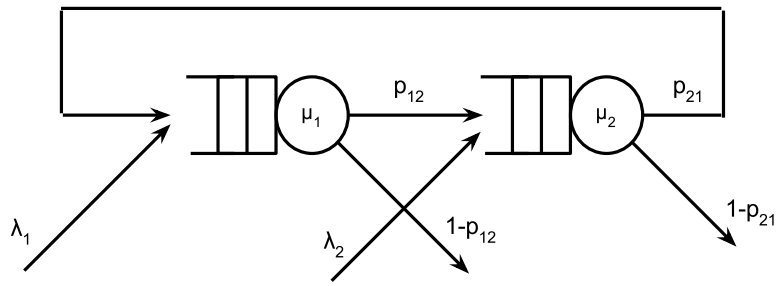

Figure 5.3: General Tandem-2 Network

The network is represented in MATLAB code as given in Figure 5.4 where the last line denotes the cooperation between the two queues.

```
P(n) = (e1, lambda1).P(n+1) for n >= 0
P(n) = (a1, infinity).P(n+1) for n >= 0
P(n) = (d1, (1-p12)*mu1).P(n-1) for n > 0
P(n) = (a2, p12*mu1).P(n-1) for n > 0
Q(n) = (e2, lambda2).Q(n+1) for n >= 0
Q(n) = (a2, infinity).Q(n+1) for n >= 0
Q(n) = (d2, (1-p21)*mu2).Q(n-1) for n > 0
Q(n) = (a1, p21*mu2).Q(n-1) for n > 0

P(0) with Q(0) over {a1, a2}
```

Figure 5.4: MATLAB Code for General Tandem-2 Network

On parsing the input, the program produces the structure R_k, where R_1 (or process P) is shown in Figure 5.5.

```
>> r(1)
definitions:
{[5x1 containers.Map]  [5x1 containers.Map]
[5x1 containers.Map]  [5x1 containers.Map]}
activeLabels: {'a2'  'd1'  'e1'}
passiveLabels: {'a1'}
```

Figure 5.5: R_1 for General Tandem-2 Network

As input in Figure 5.4, the process P has four definitions, three active actions - $\{a2, d1, e1\}$ and one passive action - $\{a1\}$. Each description of the process is stored as a map with process descriptors as keys. Running the reversed rate calculation, we get solutions as shown in Figure 5.6.

```
Reversed rate for passive action a1:
x_a1 = p21*(lambda2 + x_a2)

Reversed rate for passive action a2:
x_a2 = p12*(lambda1 + x_a1)
```

Figure 5.6: Passive rates for General Tandem-2 Network

Comparing the rates with the ones in the paper [3], we verify that they are as expected. As with the first example, we know that all the three conditions of RCAT are satisfied and this is verified by the program.

5.3 G-Networks

G-Networks as detailed in section 2.5.3, are queuing networks with negative customers. A G-network node has two customers - positive which behave as standard customers in an M/M/1 queue, and negative which are Poisson arrivals that remove, or kill, customers in the queue when it is not empty.

Consider a G-network with two nodes (see Figure 5.7), with respective positive/ negative external arrival rates λ_1, λ_2/ Λ_1, Λ_2 (corresponding to λ and Λ in Figure 5.8), with respective service rates μ_1, μ_2 and with respective positive / negative routing probabilities p_{12}, p_{21} / $(1 - p_{12}), (1 - p_{21})$. This network is represented in MATLAB code as given in Figure 5.8 where the last line denotes the cooperation between the two queues.

Since the two queues in the G-network are M/M/1 queues, every node i in isolation has a steady state probability of local state n of $\pi(n) = (1 - \rho_i)\rho_i^n$ where $\rho_i = \frac{\lambda_i + \sum_k x_{ki}}{\mu_i + \Lambda_i + \sum_k x'_{ki}}$ and $\lambda_i + \sum_k x_{ki}$ and $\mu_i + \Lambda_i + \sum_k x'_{ki}$ is the arrival rate

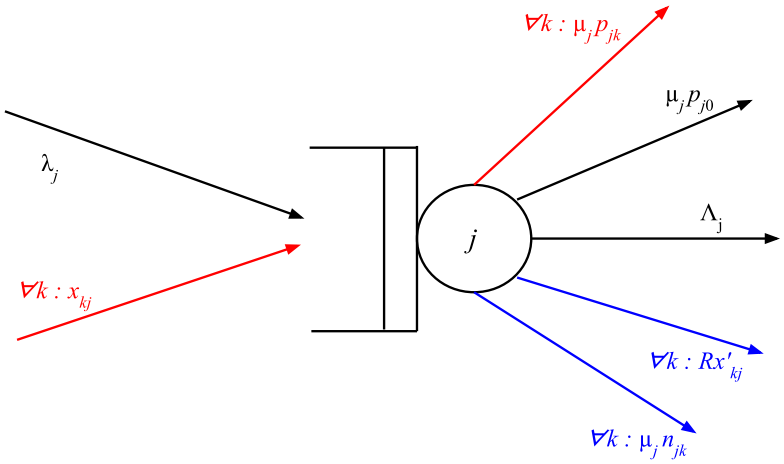

Figure 5.7: A G-Queue j within a network

and service rate at node i respectively. Since the nodes are M/M/1 queues, we can just run the reversed rate calculation of the implemented RCATscript which generates solutions given in Figure 5.9.

```
Reversed rate for passive action a21:
x_a21 = (mu2*p21*(lambda2 + x_a12))/
(bigLambda2 + mu2 + 2*x_b12)

Reversed rate for passive action b21:
x_b21 = -(mu2*(lambda2 + x_a12)*(p21 - 1))/
(bigLambda2 + mu2 + 2*x_b12)

Reversed rate for passive action a12:
x_a12 = (mu1*p12*(lambda1 + x_a21))/
(bigLambda1 + mu1 + 2*x_b21)

Reversed rate for passive action b12:
x_b12 = -(mu1*(lambda1 + x_a21)*(p12 - 1))/
(bigLambda1 + mu1 + 2*x_b21)
```

Figure 5.9: Passive rates for a Two Node G-Network

The correctness of the steady state probability calculation and reversed rate calculation is established on comparing the results with the same example in paper [4]. The reversed rates as calculated in the paper are given below for reference.

$$x_{akj} = \frac{p_{kj}\mu_k(\lambda_k + \sum_j x_{ajk})}{\mu_k + \Lambda_k + \sum_j x_{bjk}}$$

$$x_{bkj} = \frac{(1 - p_{kj})\mu_k(\lambda_k + \sum_j x_{ajk})}{\mu_k + \Lambda_k + \sum_j x_{bjk}}$$

```
1  P(n) = (e1, lambda1).P(n+1) for n >= 0
2  P(n) = (f1, bigLambda1).P(n-1) for n > 0
3  P(n) = (a21, infinity).P(n+1) for n >= 0
4  P(n) = (b21, infinity).P(n-1) for n > 0
5  P(n) = (b21, infinity).P(n) for n = 0
6  P(n) = (a12, p12*mu1).P(n-1) for n > 0
7  P(n) = (b12, (1-p12)*mu1).P(n-1) for n > 0
8
9  Q(n) = (e2, lambda2).Q(n+1) for n >= 0
10 Q(n) = (f2, bigLambda2).Q(n-1) for n > 0
11 Q(n) = (a12, infinity).Q(n+1) for n >= 0
12 Q(n) = (b12, infinity).Q(n-1) for n > 0
13 Q(n) = (b12, infinity).Q(n) for n = 0
14 Q(n) = (a21, p21*mu2).Q(n-1) for n > 0
15 Q(n) = (b21, (1-p21)*mu2).Q(n-1) for n > 0
16
17 P(0) with Q(0) over {a12, a21, b12, b21}
```

Figure 5.8: MATLAB Code for a Two Node G-Network

where x_{aij} and x_{bij} are the passive action rates corresponding to positive and negative internal arrivals respectively, a_{ij} and b_{ij}.

To ensure that the RCAT first condition holds in this network, we add invisible transitions (line 5 and 12 in Figure 5.8) to ensure negative arrivals have no effect on an empty queue. If we were to run the RCATscript omitting lines 5 and 12 from the PEPA description in Figure 5.8, we would get a condition violation error shown in Figure 5.10.

```
Error using checkFirstRcatCondition (line 10)
RCAT First Condition is violated.
If you think this is an error,
then check the pepa description and try again.

Error in RCATscript (line 31)
checkFirstRcatCondition( r, stateSpace );
```

Figure 5.10: RCAT condition Violation for a Two Node G-Network

Thus we can see that the first two RCAT conditions are satisfied from the PEPA description. Since the passive actions in the reversed process do not have multiple transitions, we conclude that the third condition of RCAT holds for this network. As with previous examples, all the three conditions of RCAT are satisfied by the program and notified to the user.

5.4 Three Node Jackson Network

This example is taken from paper [11]. Consider a three queue Jackson network, with external arrival rates $\lambda_1, \lambda_2, \lambda_3$ and service rates μ_1, μ_2, μ_3 respectively, routing probability p_{ij} from queue i to queue j, $i, j \in \{1, 2, 3\}$ and where customers leave the network from node i with probability $1 - \sum_{j \neq i} p_{ij}$. This network is represented in MATLAB code as given in Figure 5.11 where the last two lines denote the cooperation between the two queues. The three queues given as input are $P1, P2, P3$.

```
P1(n) = (e1, lambda1).P1(n+1) for n >= 0
P1(n) = (a21, infinity).P1(n+1) for n >= 0
P1(n) = (a31, infinity).P1(n+1) for n >= 0
P1(n) = (d1, (1-(p12+p13))*mu1).P1(n-1) for n > 0
P1(n) = (a12, p12*mu1).P1(n-1) for n > 0
P1(n) = (a13, p13*mu1).P1(n-1) for n > 0

P2(n) = (e2, lambda2).P2(n+1) for n >= 0
P2(n) = (a12, infinity).P2(n+1) for n >= 0
P2(n) = (a32, infinity).P2(n+1) for n >= 0
P2(n) = (d2, (1-(p21+p23))*mu2).P2(n-1) for n > 0
P2(n) = (a21, p21*mu2).P2(n-1) for n > 0
P2(n) = (a23, p23*mu2).P2(n-1) for n > 0

P3(n) = (e3, lambda3).P3(n+1) for n >= 0
P3(n) = (a13, infinity).P3(n+1) for n >= 0
P3(n) = (a23, infinity).P3(n+1) for n >= 0
P3(n) = (d3, (1-(p31+p32))*mu3).P3(n-1) for n > 0
P3(n) = (a31, p31*mu3).P3(n-1) for n > 0
P3(n) = (a32, p32*mu3).P3(n-1) for n > 0

P1(0) with P2(0) with P3(0) over
{a21, a31, a12, a32, a13, a23}
```

Figure 5.11: MATLAB Code for Three Node Jackson Network

The process for running RCAT on a system model for $k > 2$ (that is for multiple agents) is the same as running RCAT on only two compound agents. We represent the multiple processes cooperating with keyword 'with' and represent the actions the processes are cooperating on in parentheses (see last two lines of Figure 5.11).

On parsing the input, the program produces the structure R_k with three nodes as shown in Figure 5.12.

```
r = 1x3 struct array with fields:
      definitions
      activeLabels
      passiveLabels
```

Figure 5.12: R_k for Three Node Jackson Network

Running the reversed rate calculation, we get solutions as shown in Figure 5.13

```
Reversed rate for passive action a21: x_a21 =
p21*(lambda2 + x_a12 + x_a32)

Reversed rate for passive action a31: x_a31 =
p31*(lambda3 + x_a13 + x_a23)

Reversed rate for passive action a12: x_a12 =
p12*(lambda1 + x_a21 + x_a31)

Reversed rate for passive action a32: x_a32 =
p32*(lambda3 + x_a13 + x_a23)

Reversed rate for passive action a13: x_a13 =
p13*(lambda1 + x_a21 + x_a31)

Reversed rate for passive action a23: x_a23 =
p23*(lambda2 + x_a12 + x_a32)
```

Figure 5.13: Passive rates for Three Node Jackson Network

Comparing the rates with the ones in the paper [11], we verify that they are as expected. As with all the previous examples, we know that all the three conditions of RCAT are satisfied and this is verified by the program.

5.5 Stochastic Petri Nets

All examples of Stochastic Petri nets (SPNs) used for testing are taken from the paper [13]. The results in that paper have been derived by hand. We look at two SPNs for evaluation, one with two BBs and one with three BBs; thus showing that the implementation can find a product form for an SPN with several BBs.

5.5.1 Simple SPN composed of two building blocks

Figure 4.8 gives an example of a simple SPN with two building blocks (BBs) with dotted lines showing synchronising transitions. It is specified as input

in MATLAB as shown in Figure 4.10. On running `RCATscriptForSPN` on the SPN, a BB structure is created as expected with two BBs as shown in Figure 5.14.

```
1x2 struct array with fields:
  places
  inputs
  outputs
```

Figure 5.14: Building Block Structure

The program then generates rate equations and product form conditions as shown in Figure 5.15 lines 1-3 and lines 5-10 respectively. These are verified by the results in the paper : [13]. Observing BB_2 in Figure 4.8 we can see that, since the number of transitions in greater than the number of places, BB_2 does require additional conditions for a product form solution. This is indicated by the program on line 5-6.

```
1  Printing rate equations:
2  Rate equation for i_t23: x_i_t23 = lambda5
3  Rate equation for i_t45: x_i_t45 = lambda12 + x_i_t23
4
5  Product form is subject to following
6  conditions being fulfilled for BB 2:
7  rho4 = lambda4 / mu4
8  rho45 = x_i_t45 / mu45
9  rho45 = rho4 * rho5
10 rho5 = lambda5 / mu5
```

Figure 5.15: Rate equations and conditions for product form result of SPN in Figure 4.8

Solving the equations generated in Figure 5.15, a product form result can be obtained easily.

5.5.2 SPN composed of three building blocks

Figure 5.16 shows a SPN structure and its decomposed BBs. Thus the SPN considered is composed of three building blocks. We can deduce their connections based on the places in each BB. All input transitions of all BBs are unknown (passive); there are five unknown input transition rates. The service rates (output transition rates) for all three BBs are given as χ_i ,$1 \leq i \leq 5$.

We specify the building blocks as input in MATLAB as shown in Figure 5.17 and run the program using function `RCATscriptForSPN`. We deduce the synchronising transitions by observing the structure of the decomposed BBs and the SPN structure.

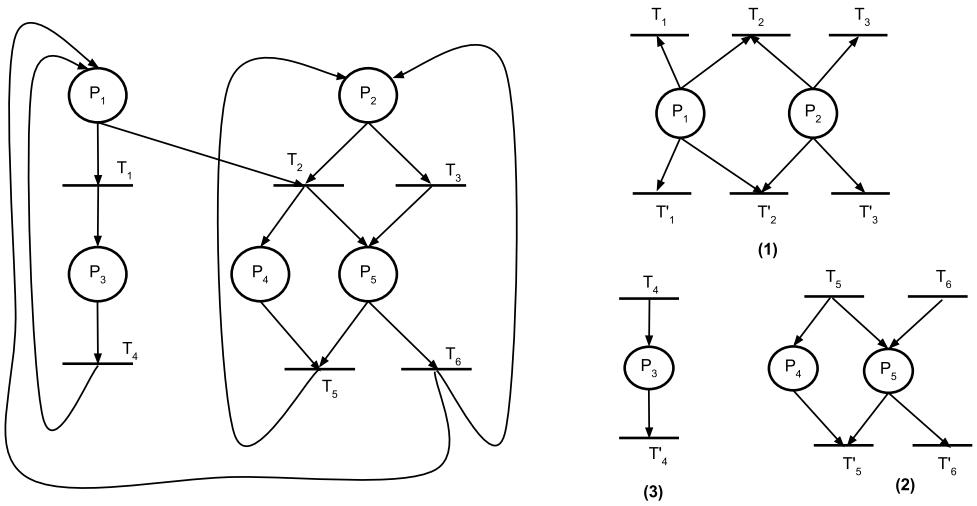

Figure 5.16: SPN with 3 BBs and decomposition of the SPN into BBs 1–3

```
listOfBBs ={
‘{P1, P2}, {i_t1, i_t2, i_t3}, {infinity,infinity,infinity},
 {o_t1, o_t2, o_t3}, {chi1, chi2, chi3}’,
‘{P4, P5}, {i_t5, i_t6}, {infinity, infinity}, {o_t5, o_t6},
 {chi5, chi6}’,
‘{P3}, {i_t4}, {infinity}, {o_t4}, {chi4}’ }

connectionString = ‘{o_t1 = 1 to i_t4} ; {o_t3 = 1 to i_t6};
{o_t2 = 1 to i_t5}; {o_t5 = 1 to i_t3}; {o_t6 = 1 to i_t2};
{o_t4 = 1 to i_t1}’

> RCATscriptForSPN( listOfBBs, connectionString )
```

Figure 5.17: Running RCAT on a SPN in Fig. 5.16

The program then generates rate equations and product form conditions as shown in Figure 5.18, lines 1-7 and lines 9-14 respectively. These are verified by the results in the paper : [13]. Observing BB_1 in Figure 5.18 we can see that, since the number of transitions in greater than the number of places, BB_1 does require additional conditions for a product form solution. This is indicated by the program on line 9-10.

```
1  Printing rate equations:
2  Rate equation for i_t1: x_i_t1 = x_i_t4
3  Rate equation for i_t2: x_i_t2 = x_i_t6
4  Rate equation for i_t3: x_i_t3 = x_i_t5
5  Rate equation for i_t4: x_i_t4 = x_i_t1
6  Rate equation for i_t5: x_i_t5 = x_i_t2
7  Rate equation for i_t6: x_i_t6 = x_i_t3
8
9  Product form is subject to following
10 conditions being fulfilled for BB 1:
11 rho1 = x_i_t1 / chi1
12 rho2 = x_i_t2 / chi2
13 rho2 = rho1 * rho3
14 rho3 = x_i_t3 / chi3
```

Figure 5.18: Rate equations and conditions for product form result of SPN in Figure 5.16

Solving the equations generated in Figure 5.18, a product form result can be obtained easily.

5.6 Strengths and Weaknesses

The strengths of the work accomplished in this project include:

- Automation of the RCAT and Multiple Agent RCAT theorem using the generic algorithm detailed in section 2.6.2.3. Due to the automation, complex queuing models with multiple synchronising processes can be analysed using RCAT with ease and their (new) product form solutions can be generated.
- Automation of product form construction for Stochastic Petri Nets (SPNs). Due to this automation, complex and large SPNs can be analysed with ease as opposed to a tedious manual process.
- Parser for agents (processes) specified in PEPA, a type of Stochastic Process Algebra. Due to the modularised code, the parser can be reused in applications, where processes are described using the same syntax and in any extensions to the RCAT.
- Parser for Stochastic Petri Nets (SPNs). A new formalism is introduced for SPNs and a parser has been implemented for the same.
- An easy command line API for product form construction of both queuing models and SPNs, which ensures that the user does not require in depth knowledge of MATLAB.

Although the majority of the project was a success, there are some limitations, as listed below:

- The implementation of RCAT and MARCAT in this project accommodates only networks composed of M/M/1 queues; so the steady state probabilities calculation automated by this implementation is applicable to only M/M/1 queues which limits the queuing models which the software can be run over.

- The implementation of RCAT for SPNs assumes the knowledge of all Building Blocks (BBs) which the given SPN is composed off. This is a disadvantage, as if the SPN is large and complex then finding BBs manually can be tedious.

- We had initially planned to provide a Graphical User Interface (GUI) for the implementation but due to time constraints it was not realised. The implementation will benefit from a GUI and due to the straightforward API it can be easily integrated with the current implementation of RCAT.

Chapter 6

Conclusions

To conclude this report, we summarise our achievements in terms of what we have learnt and consider if we have met our objectives. The goal of the project was to automate the construction of product forms in stochastic models. Using MATLAB, we have built a software system which successfully implements the generic algorithm for RCAT and MARCAT, thus achieving automatic construction of rate equations used to derive product forms for queuing models. We have as a part of the implementation developed a parser for queuing models defined in PEPA syntax. On correctly parsing the multiple process descriptions, the software has been able to generate rate equations for synchronising actions. The solutions of these equations are then used to calculate the marginal probabilities of process nodes which are then used in product form construction of queuing models.

Furthermore, we have also automated the rate equation generation (leading to product form construction) for Stochastic Petri Nets (SPNs) composed of Building Blocks in product form. A formalism has been provided for BBs and SPNs and a parser has been implemented for the same. In addition to generating rate equations, we have also provided the functionality to check whether the conditions of the BB are in product form. While this implementation requires the knowledge of all BBs which a given SPN is composed of to be known, it does provide a scope for adding a functionality where this will not be necessary. Both the implementations have been tested for a wide variety of queuing models including G-queues (queues with negative customers) and relatively large SPNs, as detailed in the Evaluation section. The software has been made with a hope to provide the first step towards mechanically analysing complex networks composed of both queuing models and SPNs and thus leading to the discovery of new product form solutions.

6.1 Future Work

6.1.1 Chains of interactions between queues

As mentioned in section 4.3, a new algorithm has been defined in [14] to generate a system of rate equations for PITs or Propagation of Instantaneous Transitions. Chains of instantaneous state changes are caused by models with

synchronisations defined as propagating instantaneous signals, for example, negative customers can move a customer from a non-empty queue to another queue chosen probabilistically thus triggering two queues (or constituent process) to change their states simultaneously. These chains are modelled as PITs to specify the composed models as pairwise synchronisations and thus derive product-forms by an iterative application of RCAT. Section 4.3 details how the current implementation can be extended to implement this functionality. As future work, RCAT for PITs will be extremely beneficial as it would extend the usefulness of the software to find rare product forms in chains of queues.

6.1.2 Automating identifying the BBs of a SPN

As mentioned before, the current implementation of RCAT for SPNs assumes the knowledge of all BBs which the given SPN is explicitly composed off. SPNs are typically large and complex which makes manually finding BBs tedious. An algorithm to automate the process is detailed in the paper [13] and can be implemented as an extension to the current implementation. The algorithm can be extended to directly produce programmable input, so the user does not need to type out verbose BB input and will directly get the generated rate equations.

6.1.3 M/M/1

The implementation of RCAT and MARCAT in this project accommodates only networks composed of M/M/1 queues; so the steady state probabilities calculation automated by this implementation is applicable to only M/M/1 queues. The implementation would benefit greatly if this restriction were removed and would thus be able to analyse queuing models which go beyond M/M/1 queues. As the program logic is loosely coupled, adding this functionality is possible without drastic changes to current implementation.

Bibliography

[1] Harrison, Peter G. and Patel, Naresh M. "Performance Modelling of Communication Networks and Computer Architectures." Addison-Wesley, 1992.

[2] Stewart, William, J. "Probability, Markov Chains, Queues and Simulation: The mathematical basis of performance modelling." Princeton University Press, 2009.

[3] Harrison, Peter G. "Turning back time in Markovian process algebra." Theoretical Computer Science, vol. 290, pp. 1947–1986, 2003.

[4] Harrison, Peter G. "Turning Back TimeWhat Impact on Performance?." The Computer Journal, vol. 53, no. 6, pp. 860–868, 2010.

[5] Harrison, Peter G., Casale, Giuliano. and Bradley, Jeremy, T. "Course 436 : Performance Analysis," Imperial College London, 2010.

[6] Bradley, Jeremy, T. "RCAT: From PEPA to Product form." Technical Report, vol. 2007, Issue 2, pp.1–8, March 2007.

[7] Gelenbe, Erol. "Product-Form Queueing Networks with Negative and Positive Customers." Journal of Applied Probability, Vol. 28, No. 3, pp. 656-663, 1991.

[8] Hillston, Jane. "A Compositional Approach to Performance Modelling." New York, NY, USA: Cambridge University Press, 1996.

[9] Harrison, Peter G., Lladó, Catalina M. and Puigjaner, Ramón. "A unified approach to modelling the performance of concurrent systems." Simulation Modelling Practice and Theory 17 , no. 9 (2009): 1445-1456.

[10] Harrison, Peter G. "Reversed processes, product forms and a non-product form." Linear Algebra and Its Applications, Volume 386, pp.359–381, 2004.

[11] Harrison, Peter G., Lee, Ting. "Reversed Processes of Multiple Agent Cooperations." 19th UK Performance Engineering Workshop, pp.257–265, 2003.

[12] Harrison, Peter G., Lee, Ting T. "Separable equilibrium state probabilities via time reversal in Markovian process algebra." Theoretical Computer Science, vol 346, pp.161–182, 2005

[13] Balsamo, Simonetta., Harrison, Peter G., Marin, Andrea. "Methodological construction of product-form stochastic Petri nets for performance evaluation." The Journal of Systems and Software 85 (2012) 1520–1539.

[14] Harrison, Peter G., Marin, Andrea. "Product-forms in multi-way synchronisations". The Computer Journal.

Appendix A

MATLAB Code for Unit Tests

A.1 Unit Test for registering processes

```
1  function testRegisterProcess()
2
3  % Setting up the process for testing
4    syms lambda n x_a;
5    registeredProcesses = containers.Map();
6    activeActionLabels = containers.Map();
7    passiveActionLabels = containers.Map();
8    registerProcess( registeredProcesses, activeActionLabels,...
9    passiveActionLabels, 'P(n) = (e, lambda).P(n+1) for n >= 0' );
10   P = registeredProcesses( 'P' );
11 % Retrieve process registered
12   P = P{1};
13
14 % Test if process descriptors have been parsed correctly
15   assertEqual( registeredProcesses.length(), 1 );
16   assertEqual( P( 'actionName' ), 'e' );
17   assertEqual( P( 'actionRate' ), lambda )
18   assertEqual( P( 'transitionFromState' ), eval('n') );
19   assertTrue( isequal( P( 'transitionToState' ), n+1 ) );
20
21 % Test the domain function by giving it actual values
22   domain = P( 'domain' );
23   assertTrue( isequal( 0, domain(1) ) );
24   assertTrue( isequal( Inf, domain(2) ) );
25
26 % Add Passive process to test for parsing and relabelling rates
27   registerProcess( registeredProcesses, activeActionLabels,...
28   passiveActionLabels, 'Q(n) = (a, infinity).Q(n+1) for n >= 0' );
29   assertEqual( registeredProcesses.length(), 2 );
30   Q = registeredProcesses( 'Q' );
31   Q = Q{1};
32
33 % Test passive rate relabelling
34   assertEqual( Q( 'actionRate' ), x_a );
35
36 % the input has one process with active action type and one with
37 % passive action type. Thus the foll. assertions
38   assertEqual( activeActionLabels.length(), 1 );
39   assertEqual( activeActionLabels('P'), { P('actionName') } );
40   assertEqual( passiveActionLabels.length(), 1 );
```

```
41    assertEqual( passiveActionLabels('Q'), { Q('actionName') } );
42
43  % Test if input validates correctly i.e. check exceptions thrown
44    assertExceptionThrown( @() registerProcess( ...
45    registeredProcesses, activeActionLabels,...
46    passiveActionLabels, 'Q(n) = a, infinity)   Q(n+1)  n >= 0' ), ...
47    'RCATscript:InvalidInputParsedRegisterProcess' );
48
49    assertExceptionThrown( @() registerProcess(...
50    registeredProcesses, activeActionLabels,...
51    passiveActionLabels, '' ), 'RCATscript:InvalidInputRegisterProcess' );
52
53    assertExceptionThrown( @() registerProcess(...
54    registeredProcesses, activeActionLabels,...
55    passiveActionLabels, 'Q(n) = (2e, infinity).Q(n+1) for n >= 0' ),...
56    'RCATscript:NumericActionLabel' );
57
58  end
```

This program listing unit tests the part of the implementation to do with parsing. It checks if a PEPA process is correctly parsed using various assertions. The comments on the listing (lines starting with '%') provide context to the assertions. Different parts of the program have been unit tested with a similar approach.

A.2 Unit test for string to expression generation

```
1   function testStringToMatlabExpr()
2
3   % Declare variables symbolic for test case set up
4     syms lambda1 mu1 lambda_1 mu_1;
5
6   % Simple case
7     assertTrue( isequal( mu_1, stringToMatlabExpr('mu_1') ) );
8
9   % Non-trivial cases, examples added with spaces in expr
10  % which should be ignored by the function
11    assertTrue( isequal(( 1 - lambda1 ) * mu1, ...
12    stringToMatlabExpr('(1-lambda1)*mu1') ) );
13
14    assertTrue( isequal( ( 1 - lambda1 ) * mu1, ...
15    stringToMatlabExpr('( 1 - lambda1 ) * mu1') ) );
16
17    assertTrue( isequal( ( 1 - lambda_1 ) * mu_1, ...
18    stringToMatlabExpr('(1-lambda_1)*mu_1') ) );
19
20  % Test if maths expressions are evaluated correctly
21    assertTrue( isequal(7404, stringToMatlabExpr('(1234)* 6') ) );
22
23  % Test if wrong output is caught correctly
24      assertExceptionThrown( @() stringToMatlabExpr('(1-)*mu1'),
25    'RCATscript:InvalidStringToMatlabExpr' )
26  end
```

This program listing unit tests the part of the implementation to do with converting strings to MATLAB symbolic expressions. It is relatively simple and shows that the program stringToMatlabExpr has been tested over various

inputs including an erroneous input.

A.3 Running Unit Tests

The instructions below assume that the xUnit Framework has been installed in MATLAB and is ready to run.

```
% go to the directory containing the tests
>> cd tests/

% The run the undermentioned command
>> runtests

% Test output looks as follows
Test suite: /Users/.../Code/tests
Starting test run with test cases.
.....PASSED in 11.191 seconds.
```