Analysing Clocked Process Algebras via Stochastic Petri Nets

Wallace Wai Kean Wong

Bachelor of Engineering Project Report
3rd July 2004

Dr A.J. Field
Dr J.T. Bradley

Department of Computing, Imperial College London, United Kingdom
Abstract

Correctness and performance are two of the most important engineering issues in the development of complex software. Clocked Process Algebras (PA) such as SFSP (stochastic Finite State Processes) offer a means for systematic, hierarchical modelling of complex systems for determining both aspects of a system. SFSP is a process algebra supported by the LTSA (Labelled Transition System Analyser) tool. Traditionally, clocked PAs are solved by mapping them to Generalised Semi-Markov Processes (GSMPs) or Stochastic Timed Automata (STAs); these are then “solved” by discrete-event simulation.

This report will focus on solving clocked PAs numerically, by translating the model to a semi-Markov stochastic Petri net (SM-SPN); the net can then be solved by DNAmaca, a SM-SPN analyser that can generate and solve both Markov and semi-Markov processes in large state spaces. A translation scheme for converting SFSP into SM-SPNs is then presented. The translation scheme and integration with DNAmaca has been implemented as an LTSA plug-in. This tool, which also allows for the net to be exported to be analysed with other Petri net tools such as PIPE and Medusa, will also be presented.
Acknowledgements

I would like to express my thanks and gratitude to:

- My supervisor, Dr. Tony Field, for his invaluable guidance and patience, and for spending many hours with me when I encountered numerous problems in the course of the project.

- My second marker, Dr. Jeremy Bradley, for his input regarding the translation of a process algebra to a Petri net, and his encouragement and friendliness.

- My personal tutor, Dr. William Knottenbelt, for a tutorial on Petri nets, a textbook by Prof. P.S. Kritzinger, and his help on DNAmaca.

- Nick Dingle, for his patience and enormous help with DNAmaca.
Contents

1 Introduction ................................................. 1
  1.1 Motivation .............................................. 1
  1.2 Objectives .............................................. 3
  1.3 Report outline ......................................... 3

2 Background ................................................. 5
  2.1 Introduction ............................................ 5
  2.2 Solution methods ...................................... 6
    2.2.1 Analytical .......................................... 6
    2.2.2 Simulation .......................................... 7
    2.2.3 Numerical ........................................... 8
  2.3 Markov Chains ....................................... 11
    2.3.1 Discrete-Time Markov Chain (DTMC) .............. 11
    2.3.2 Continuous Time Markov Chains (CTMC) .......... 13
    2.3.3 Semi-Markov Processes (SMP) ..................... 14
    2.3.4 Generalised semi-Markov Processes (GSMP) ....... 15
    2.3.5 Markov Chains lack compositionality ............ 15
  2.4 Petri nets ............................................ 16
    2.4.1 Properties of Petri nets ......................... 17
    2.4.2 Stochastic Petri nets ............................. 19
    2.4.3 Generalised Stochastic Petri nets ............... 20
    2.4.4 Semi-Markov Stochastic Petri nets ............. 20
    2.4.5 Queueing Petri nets ............................. 21
    2.4.6 Pros and Cons of the Petri net formalism ...... 22
  2.5 Process Algebras .................................... 22

v
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Stochastic Process Algebras</td>
<td>24</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Markovian SPAs</td>
<td>24</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Non-Markovian SPAs</td>
<td>26</td>
</tr>
<tr>
<td>2.6.3</td>
<td>SFSP</td>
<td>28</td>
</tr>
<tr>
<td>2.7</td>
<td>Specifying performance measures</td>
<td>29</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Types of performance measures</td>
<td>29</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Logic based approach</td>
<td>30</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Process algebra based approach</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Survey of existing tools</td>
<td>33</td>
</tr>
<tr>
<td>2.8.1</td>
<td>SPA tools</td>
<td>33</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Petri-net tools</td>
<td>34</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Queuing network tools</td>
<td>35</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Multiple formalisms</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>LTSA and SFSP</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>Translating SFSP to a Petri net</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Grammar of SFSP−</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Informal semantics of SFSP−</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Clocks</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Formal semantics of SFSP−</td>
<td>42</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Mapping SFSP− to SA</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Equivalence of SFSP− and Petri nets</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Single process</td>
<td>46</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Multiple processes</td>
<td>47</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Clocks</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>Translation scheme of SFSP− to a Petri net</td>
<td>52</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Top level translation</td>
<td>53</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Translating processes</td>
<td>54</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Translating prefixes</td>
<td>56</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Translating clocks</td>
<td>57</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Examples</td>
<td>58</td>
</tr>
<tr>
<td>3.6</td>
<td>Alphabet extensions, hiding, relabling and priority</td>
<td>61</td>
</tr>
</tbody>
</table>
3.6.1 Hiding .......................................................... 61
3.6.2 Relabeling ....................................................... 61
3.6.3 Alphabet extensions ........................................... 62
3.6.4 Priority .......................................................... 62
3.7 Performance measures ............................................ 64
3.7.1 Translating performance measures ......................... 64
3.8 Non Markovian processes ....................................... 67

4 Handling probability .............................................. 69
4.1 New probabilistic model ......................................... 69
4.1.1 What was wrong? ........................................... 69
4.1.2 What should it be? .......................................... 70
4.2 Formalising SFSP with probability .............................. 71
4.2.1 Why do we need a new transition system? ................ 71
4.2.2 Clocked Generative Probabilistic Transition System ....... 72
4.2.3 Mapping SFSP to CGPTS .................................. 72
4.2.4 Semantics of CGPTS ....................................... 74
4.3 Translating probabilities ......................................... 77

5 Design and Implementation ........................................ 83
5.1 Implementing the conversion .................................... 83
5.1.1 LTSA components .......................................... 84
5.1.2 Petri net components ....................................... 85
5.1.3 Conversion scheme ....................................... 87
5.2 Exporting the Petri net .......................................... 91
5.2.1 Visitor design pattern ..................................... 91
5.2.2 PNML .......................................................... 92
5.2.3 DNAmaca ..................................................... 93
5.3 Getting and displaying results .................................. 94
5.3.1 User Interface (plugin component) ......................... 94
5.3.2 Integrating with DNAmaca (Runtime.exec, threads) ....... 97
5.3.3 Graphs .......................................................... 98
5.4 Tools used .......................................................... 98
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Evaluation</td>
</tr>
<tr>
<td>6.1 Verification of Petri net generation</td>
</tr>
<tr>
<td>6.1.1 Composition</td>
</tr>
<tr>
<td>6.1.2 Alphabet extensions</td>
</tr>
<tr>
<td>6.1.3 Probability</td>
</tr>
<tr>
<td>6.1.4 Clocks</td>
</tr>
<tr>
<td>6.1.5 Performance measures</td>
</tr>
<tr>
<td>6.2 Verification of translation scheme through testing</td>
</tr>
<tr>
<td>6.2.1 Simple clock</td>
</tr>
<tr>
<td>6.2.2 M/M/1/K</td>
</tr>
<tr>
<td>6.2.3 Tagged queue</td>
</tr>
<tr>
<td>6.2.4 Case study: Engineering workshop [12]</td>
</tr>
<tr>
<td>6.3 Qualitative evaluation</td>
</tr>
<tr>
<td>6.3.1 Strengths</td>
</tr>
<tr>
<td>6.3.2 Weaknesses</td>
</tr>
<tr>
<td>7 Conclusion</td>
</tr>
<tr>
<td>7.1 Discussion</td>
</tr>
<tr>
<td>7.1.1 Contributions</td>
</tr>
<tr>
<td>7.1.2 Challenges</td>
</tr>
<tr>
<td>7.2 Future work</td>
</tr>
<tr>
<td>7.2.1 Specification of performance measures</td>
</tr>
<tr>
<td>7.2.2 Fixing probability extraction</td>
</tr>
<tr>
<td>7.2.3 Packaging/ Re-engineering DNAmaca</td>
</tr>
<tr>
<td>7.2.4 Proof of correctness of translation scheme</td>
</tr>
<tr>
<td>A Acronyms</td>
</tr>
<tr>
<td>Bibliography</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Correctness and performance are two of the most important engineering issues in the development of complex software. Correctness describes the *qualitative* aspects of a system, such as liveness, safety, boundedness and fairness. Performance describes the *quantitative*, dynamic, time-dependent behaviour of systems, such as response time, system uptime and throughput.

Though they have historically been treated as two separate issues, the need for integrated modelling techniques was already recognised in the 1970s. The most successful examples of such integrated modelling techniques[42] are stochastic Petri nets [6], stochastic graph models and stochastic automata networks.

There are several tools available which allow the performance analyst to model a system using one of the above formalisms, and derive both qualitative and quantitative performance measures. Several of these tools are discussed in Chapter 2. DNAmaCa, one of such tools, allows the user to specify a model in a Stochastic Petri Net. It is then able to generate a Markov chain using a fast probabilistic state space generation technique [52] and evaluate performance measures numerically by solving the generator matrix.

Although these modelling formalisms allow the evaluation of both correctness and performance, they are an art mastered only by a small group of specialists. Process algebras offer means for systematic, hierarchical modelling of complex systems, but they are only used for *qualitative* analysis (correctness) because they lack time and probabilistic information. With the relatively recent emergence of the stochastic extensions of process algebras, both *qualitative* and *quantitative* performance evaluation can be carried out with a single, integrated modelling environment in a systematic approach.
Recently, there is also the option of deriving Stochastic Process Algebra models using UML\(^1\) diagrams [56], and such an option has been implemented in PEPA\(^2\) [36]. This allows a unified approach to the design and evaluation of a system even before its implementation.

The LTSA\(^3\) tool, and its associated process algebra FSP\(^4\), has been widely used in the teaching community all over the world\(^5\) because it is small, elegant, easy to use, and yet robust. In addition to property checking, it also provides a visualiser to view the generated Labelled Transition System, and an animator to step through the automaton. Unfortunately, FSP, as a process algebra without any dynamic, timed behaviour, is unable to model a stochastic system.

This led Ayles [2] to extend FSP to allow the programmer to annotate an FSP program with probabilities and time delays. The resulting language, SFS\(^6\)P, has great expressive power. Combined with discrete-event simulation, the extended LTSA tool is able to derive performance properties of many systems.

There is also ongoing research in translating UML diagrams of systems into FSP programs at Imperial College. Thus, the tool is able to analyse the \textit{qualitative} and \textit{quantitative} aspects of a system in a unified approach early in the design stage of system development. However, although this simulation tool is very general, it can be slow for some systems as it involves traversing potentially very large state spaces to generate performance estimates within a reasonable confidence interval.

Performance evaluation using discrete-event simulation, in general, though allowing for very large, complicated systems with different distributions (non-Markovian), usually requires very long runs to obtain results of good accuracy. Analytical analysis, on the other hand, though it is fast and accurate, allow for only highly restricted Markovian Chains. Therefore, the most promising method of performance evaluation is numerical analysis, which is more accurate than simulation, more flexible than analytical methods, and provides \textit{what if} analysis (perturbation, sensitivity, optimisation). Although numerical analysis has a problem of coping with extremely large state spaces (a result of state-space explosion), much research into mitigating the problem has been carried out in recent years. Research into reducing the state space [18, 44, 57], efficient computation [59, 49, 38] and parallel computation [50, 16, 51, 19] has served to increase the state sizes at which numerical analysis is effective.

\(^1\)Unified Modelling Language  
\(^2\)Performance Evaluation Process Algebra [25]  
\(^3\)Labelled Transition System Analyser [54]  
\(^4\)Finite State Process [54]  
\(^5\)UK - Queen’s University Belfast, University College London, University of Manchester, University of Dublin, Heriot Watt University (Edinburgh), Australia - University of Queensland, Italy - Università degli Studi di Padova, USA - Williams College, Germany - Universität Kaiserslautern, Japan - University of Tokyo, Denmark - Aalborg University, Singapore - Nanyang Technological University  
\(^6\)Stochastic Finite State Processes [2]
1.2 Objectives

The main aim of the project, therefore, is to extend the LTSA tool so that it is able to solve performance models numerically. In the larger context, such an extension will allow performance analysts yet another means to derive both qualitative and quantitative aspects of a system. The benefits of numerical methods over simulation is elaborated in Chapter 2.

1.2 Objectives

- To investigate the various methods of performance evaluation, including the modelling formalisms and evaluation methods, and the exploration of the tools available. This will be summarised in Chapter 2. This is to gain an understanding of the wider context of performance evaluation and the various alternatives in unifying the various alternatives of extending the LTSA tool.

- To formalise the semantics of SFSP in line with other Stochastic Process Algebras.

- To formulate a translation of SFSP into a stochastic Petri net so that performance measures can be evaluated numerically.

- To extend the stochastic version of the LTSA tool that implements the translation scheme, and directly interface with DNAmaca. Ideally, the specification of performance measure can be done through SFSP, or through a graphical user interface. The results should be displayed graphically, with the user having options to change the display or tweak the results.

1.3 Report outline

The report will cover the following:

- Background information on the various solution methods to derive performance measures, Markov chains, Petri nets, stochastic Process Algebras, methods of specifying performance measures, and some existing performance analysis tools (Chapter 2).

- The semantics of a SFSP without probability, SFSP, the equivalence of this grammar and Petri nets, and a translation scheme for compiling SFSP into a Petri net (Chapter 3).

- The semantics and translation of the full grammar of SFSP, including probability. (Chapter 4).
• The design and implementation of a LTSA plug-in that implements the translation scheme and interfaces with DNAmaca. Other features of this plug-in, such as the ability to export the Petri net to other tools will be discussed (Chapter 5).

• The evaluation of the plug-in and the translation scheme, and also the strengths and weaknesses of this approach (Chapter 6).

• Contributions, challenges faced, and future work (Chapter 7).
Chapter 2

Background

2.1 Introduction

Performance analysis typically involves the following steps [20]:

**Constructing a model**  The system of interest is abstracted into some high-level language, usually Process Algebras (see Section 2.5), so that the model approximates the system, and yet it is simple enough to be analysed. Alternatively, a system can be described in terms of a high-level structure, such as a Queuing Network or a Petri net (see Section 2.4).

**Determining performance measures** Measures such as mean queue length, mean waiting time, throughput, probability of loss due to limited queue length, utilisation, mean time to failure, mean time between failures and system availability can be asked about the system. There are various ways to describe these performance measures (see Section 2.7). One way is to specify these measures as a separate logic such as CSL\(^1\). An alternative is to specify these measures together with the system in the model [1, 2].

**Fixing parameters** The parameter such as the arrival rate \(\lambda\) and service rate \(\mu\) are fixed so that the system can be analysed. These parameters are either known, measured, estimated, or guessed.

**Deriving performance measures** There are three basic approaches: analytical, numerical and simulation (see Section 2.2).

The following will be covered in this chapter:

---

\(^1\)Continuous Stochastic Logic [4, 17]
The three different approaches to determining performance measures. More emphasis is placed on numerical analysis, for reasons that will be discussed.

Markov chains and processes. Markov chains are convenient mathematical formalisms for numerical analysis. There is much research into efficient algorithms for the analysis of Markov chains, even large ones of 100 million states [16]. An overview about Markov chains in Section 2.3 is presented.

Petri nets. Petri nets are high level formalisms that can also be used to model systems. In particular, we will be looking at Stochastic Petri nets and Queueing Petri nets as they can be used for quantitative performance analysis (Section 2.4).

Process Algebras and Stochastic Process Algebras (Sections 2.5 and 2.6). These are high level modelling languages that can be used to model a system. SFSP will be presented in more detail.

Methods of specifying performance measures. In particular, logic-based and process algebra based approaches [4, 17, 1, 24] will be elaborated (Section 2.7).

Several available performance modelling or analysis tools, how they work together, and the pros and cons of these tools (Section 2.8). The LTSA tool[2] tool will then be evaluated and compared with these existing tools (Section 2.9).

2.2 Solution methods

2.2.1 Analytical

Analytical methods are fast, simple and accurate. The only drawback of analytical methods is that it works for only highly restricted classes of Markov Chains, which is not feasible for most real-world systems.

Herzog [43] describes several analytical approaches to determining performance measures. Here are some of them:

State-based Approach This approach is based on the principle of using the Chapman-Kolmogorov or Steady State Equations to determine the steady state probabilities. Some examples of this approach include: Standard Approaches (MP), Phase Concept and Matrix Analytic Method, Embedded Renewal Processes
2.2. Solution methods

**Event Sequence Approach** The principle behind this approach is to explore the special sequences of events. Lindley’s Integral Equations and Method of Moments are some examples of this approach.

**Bounding Techniques** This approach considers the process under special conditions. For example, Asymptotic Bounds and Balanced System bounds.

**Fundamental Laws** This approach uses some fundamental laws to derive the performance measures. Some of these laws are: Little’s Law, Flow Equivalence Law and Work Conservation Law

**Transformation Techniques** This is based on the principle of changing the form of an equation to one which is easier to investigate. For example, the Laplace-Transform, Generating Functions, Discrete Fourier-Transform and Complex Cepstrum.

**Special Network Algorithms** The principle behind this is to utilize the special structure and system parameters. Jackson’s State Probabilities and Bunzen’s Convolution Algorithm works this way.

2.2.2 Simulation

As Herzog [43] has put it so succinctly: “An analytical model should be sought wherever possible, since it can evaluate the performance with minimal efforts and cost over a wide range of choices in the system parameters and configurations. However, even with simplifying assumptions and decompositions, the resultant analytic model is often not mathematically tractable. Then the only alternative for predicting the performance of a non existing system is simulation.”

There are many advantages of simulation over the other methods. First of all, it can handle very general, non-markovian models. Secondly, it can handle very large and complicated models, which is often mathematically intractable. Lastly, it is simple to program (some tools even construct the simulation automatically) and the results are easy to interpret.

However, simulations does have some drawbacks. First of all, good accuracy usually requires very long runs. Second, real optimization at reasonable expense is almost impossible. One will have to manually change the parameters and hope for the best. Third, it can be very expensive to run (simulation of one second real-time of an ATM-ring configuration took four hours of simulation time on a SPARC) [43]. Lastly, rare events may take forever to show up on a simulation, and even then, the results will not be accurate.
2.2.3 Numerical

Numerical solution techniques are more accurate than simulation, and more flexible than analytical techniques [7]. It can deal with reasonably large Markov Chains, and most importantly, it can provide what if analysis (perturbation, sensitivity, optimisation).

Although complexity is a limiting factor, much research into mitigating this problem has resulted in the ability to solve problems with a larger state space. Some of these methods include reducing the state space [44, 57] before computation, efficient computation [49, 38], and parallel (distributed) computation [50, 16, 51, 19].

Here is a (non-exhaustive) list of some of the numerical techniques that are available [43]. Some of them will be covered in this section.

**direct methods** principle: operate and modify matrix eg. Gaussian elimination, Grasman, Taksar, Heyman variant. Reliable and accurate for small models with up to thousands of states.

**iterative methods** principle: preservation of matrix sparsity, successive convergence to the solution. eg. Power method, Jacobi, Gauss-Seidel, SOR

**other methods** Projection, recursive, uniformisation (randomisation, jensen, transient)

Numerical techniques can be used to derive both steady state and transient state probabilities. Steady state probabilities are readily found from the solution of this equation:

\[ \pi Q = 0, \sum_{i \in \text{states}} \pi_i = 1 \]

which corresponds to the form of solving the equation \( A\pi = b \), where \( A = Q \) and \( b = 0 \).

Transient state probabilities are derived from solving this equation:

\[ \pi'(t) = \pi(t)Q \]

Although steady state probabilities are usually sufficient for most models, there are some times when the transient probability is of much importance. We might be interested in the transient state probabilities when:

- the system life-time is so short that steady-state is not reached.
- the period towards the steady state situation is of interest
2.2. Solution methods

- temporary overload periods are of interest
- non-repairable systems that are failure prone are of interest (by the time steady state is reached, they will have completely failed).

We will now describe some of the simpler solution methods

**Direct Methods**

Direct methods solve a system of equations by rewriting it in such a form that explicit expressions are obtained. It is able to derive *exact* results in a *a priori* known number of operations, given the number of states \( N \).

**Gaussian Elimination** Complexity of \( O(N^3) \). Main problem lies in its storage requirements. Although for most models the matrix will initially be sparse, the reduction procedure results in *fill-ins*, which is the generation of non-zero elements in the matrix. For large matrices, for example where \( N \) is in the order of \( 10^5 \) or \( 10^{10} \), Gaussian elimination cannot be performed.

**LU Decomposition** Has same order of complexity of \( O(N^3) \). Similar to Gaussian Elimination, LU Decomposition results in fill-ins.

**Iterative Methods**

Iterative methods do not change the involved matrices, therefore avoiding fill-ins, so that the matrices can be stored efficiently using sparse matrix methods. They do not result in an explicit solution of the system of equations, and they do not produce *exact* results. It is also not known before computation how many steps are required. In iterative methods, a simple iteration step is performed repeatedly until a desired level of accuracy is achieved.

Iterative methods, in a non-sparse implementation, is of the complexity of \( O(N^2) \). However, in some sparse matrix implementations, the complexity is of \( O(\eta) \), where \( \eta \) is the number of non-zero elements.

**The Power Method** This involves successive multiplication of the steady state probability vector \( v \) with \( P \) until convergence is reached. i.e., it applies the multiplication

\[
\begin{align*}
v &= \lim_{n \to \infty} \pi(n) = \lim_{n \to \infty} \pi(0)P^n
\end{align*}
\]
We can therefore see that the Power method is thus not very efficient, and will not discuss it further.

The Jacobi Method  The set of linear equations is rewritten in the following way:

\[
\sum_{j=1}^{N} a_{i,j} \pi_j = 0 \Rightarrow \pi_i = -\frac{1}{a_{i,i}} \left( \sum_{j \neq i} \pi_j a_{i,j} \right)
\]

and then the next estimate is computed as:

\[
\pi_i^{(k+1)} = -\frac{1}{a_{i,i}} \left( \sum_{j \neq i} \pi_j^{(k)} a_{i,j} \right)
\]

We continue to iterate until two successive estimates for \( \pi \) differ by less than some small number, \( \varepsilon \). Note that even when \( \varepsilon \) is small, it does not mean that the solution has been found. It could be that the convergence is slow. There are some other methods of checking convergence, but it is out of the scope of this report.

The Gauss-Seidel Method  Instead of using just the \( \pi^{(k)} \) values to estimate the \( \pi^{(k+1)} \) values, we can use the \((k+1)\)-th value as soon as it is computed to estimate the next value. That is, the iterative scheme is:

\[
\pi_i^{(k+1)} = -\frac{1}{a_{i,i}} \left( \sum_{j < i} \pi_j^{(k+1)} a_{i,j} + \sum_{j > i} \pi_j^{(k)} a_{i,j} \right)
\]

This scheme requires only that one probability vector be stored, compared with the Jacobi method, which requires two \((k\)-th and \((k+1)\)-th estimate). The Gauss-Seidel method is usually faster than the Jacobi method.

The SOR Method  SOR is the \textit{successive over-relaxation} method. It is an extension of the Gauss-Seidel method, in which the vector \( \pi^{(k+1)} \) is computed as the weighted average of the vector \( \pi^{(k)} \) and the vector \( \pi^{(k+1)} \) that would have been computed from the Gauss-Seidel iteration. That is, the iterative step is:

\[
\pi_i^{(k+1)} = (1 - \omega) \pi_i^{(k)} - \frac{\omega}{a_{i,i}} \left( \sum_{j < i} \pi_j^{(k+1)} a_{i,j} + \sum_{j > i} \pi_j^{(k)} a_{i,j} \right)
\]

where \( \omega \) is usually \( 0 \leq \omega \leq 2 \). When \( \omega \) is 1, we can see that it is exactly the same as the Gauss-Seidel method. When we take \( \omega > 1 \) or \( \omega < 1 \) its an \textit{over-relaxation} or \textit{under-relaxation} respectively. By choosing a proper \( \omega \), the iterative solution process can be
2.3 Markov Chains

A Markov chain is a stochastic process that satisfies the Markov property. A stochastic process is a family of random variables \( \{X(t)\} \), indexed by time parameter \( t \). The Markov property requires that the next state can be determined knowing nothing other than the current state. It is also described as a memoryless property. This property can be mathematically described as follows: Given \( t_0 < t_1 < \ldots < t_n \),

\[
P\{X(t_{n+1}) \leq x_{n+1} \mid X(t_n) \leq x_n, \ldots, X(t_0) \leq x_0\} = P\{X(t_{n+1}) \leq x_{n+1} \mid X(t_n) \leq x_n\}
\]

Markov chains are well investigated and many analytic results as well as efficient numeric analysis are known. In fact, Markov chains and stochastic process form the basis for model-based system evaluations in many areas of science and engineering. For example, in biology, to model growth and decay in populations; in physics, to model interactions between elementary particles; in chemical engineering, to model reactions between molecules or the mixing processes; in management science to model flow of commodities in logistic or flexible manufacturing systems or availability of production lines; most notably, in computing, to model system performance and dependability [40].

Without going into much detail, we describe the various types of Markov chains below. More detail on Markov chains can be found in Stewart [59] and Bause and Kritzinger [6]. These books include definitions, examples and exercises on Markov chains.

2.3.1 Discrete-Time Markov Chain (DTMC)

A DTMC is a Markov Chain where the time passes in discrete steps. That is, the time instances are denumerable and can be seen as elements of integers. We can therefore describe the Markov chain as a family of random variables \( \{X_t, t \in T\} \), where \( T = \{0, 1, \ldots, n\} \).

From the Markov property as described previously, we know that the actual time instances are not important, only their relative differences. That is:

\[
P\{X_{n+1} = i \mid X_n = j\} = P\{X_{m+1} = i \mid X_m = j\} \text{ for all } n, m \in \mathbb{N}
\]
Chapter 2. Background

The probability of going from state $j$ at time $m$ to state $k$ at time $n$ is $p_{j,k}(m,n) = P\{X_n = k \mid X_m = j\}$, but because the transition probabilities only depend on the time difference $l = n - m$, we can denote them as $p_{j,k}(l) = P\{X_{m+l} = k \mid X_m = j\}$. The one-step transition probabilities are denoted as $p_{j,k}$. Intuitively, the DTMC is totally described by the initial probabilities and the one-step probabilities $p_{j,k}$. Therefore, we can describe the entire DTMC in terms of a matrix $P$, where each entry of $P$ is a one-step probability $p_{j,k}$.

Fig. 2.1 shows an example DTMC as a state transition diagram, where the edges represent the probability of changing from one state to the other. Shown beside the state transition diagram is the matrix $P$, representing exactly the same information as the state transition diagram.

![Figure 2.1: Example of a Discrete Time Markov Chain (from Haverkort [40])](image)

The $n$-step probability is $P^n$, that is, the matrix product of itself $n$ times. From this, it can be shown that $P^{n+m} = P^m P^n$, and that equation is also known as the Chapman-Kolmogorov equation.

To determine the transient probability, that is, the probability of residence in state $j$ after $n$ steps, $\pi_j(n)$, all that is needed is to multiply the matrix $P^n$ with the initial probability. In matrix-vector notation, with $\mathbf{\pi}(n) = (\pi_0(n), \pi_1(n), \ldots)$, it is

$$\mathbf{\pi}(n) = \mathbf{\pi}(0) P^n$$

To determine the steady state probability, represented by the vector $\mathbf{v} = (\ldots, v_j, \ldots)$, where

$$v_j = \lim_{n \to \infty} \pi_j(n) = \lim_{n \to \infty} P\{X_n = j\} = \lim_{n \to \infty} \sum_{i \in \text{states}} \pi_i(0) p_{i,j}(n)$$
2.3. Markov Chains

we can solve the equation:

\[ v = \lim_{n \to \infty} \pi(n) = \lim_{n \to \infty} \pi(0)P^n = \lim_{n \to \infty} \pi(0)P^{n+1} = \left( \lim_{n \to \infty} \pi(0)P^n \right)P = \nu P \]

\[ v = \nu P \Rightarrow v(I - P) = 0 \]

2.3.2 Continuous Time Markov Chains (CTMC)

A continuous-time Markov chain is a Markov process with discrete state space and a state that may change at any time. We can therefore describe a CTMC as a stochastic process \( \{X(t), t \in T\} \) where \( T \) can come from the set \( \mathbb{R} \).

CTMC also satisfy the Markovian property, that is:

\[ P\{X(t_{n+1}) \leq x_{n+1} | X(t_n) \leq x_n, \ldots, X(t_0) \leq x_0\} = P\{X(t_{n+1}) \leq x_{n+1} | X(t_n) \leq x_n\} \]

This means that the sojourn times are (negatively) exponentially distributed. CTMCs are a well-studied class of stochastic processes, and several efficient algorithms for steady state and transient analysis are available. These will not be covered in this project report. See Stewart [59] for more details.

A CTMC can be represented by the matrix \( Q \) with \( q_{i,j}, i \neq j \) representing the transition rate between state \( i \) and state \( j \) and \( q_{i,i} = -\sum_{j \neq i} q_{i,j} \). The sojourn time in state \( i \) is exponentially distributed with parameter \( -\sum_{j \neq i} q_{i,j} = q_{i,i} \).

It can be shown that the CTMC defined above have the same properties as the DTMC, even though the DTMC, represented by \( P \), a matrix of probabilities, seem completely different from the CTMC, which is represented by \( Q \), a matrix of transition rates.

An example of a CTMC is shown in Fig. 2.2. Here, the edges represent the transition rate between two states. The transition rate is the parameter for the negative exponential distribution. Shown beside the state transition diagram is the matrix \( Q \), which, together with the initial probabilities, completely describe the CTMC.

The transient state probability can be determined by solving a set of linear differential equations (with respect to the time). It is represented as:

\[ \pi'_i(t) = \lim_{h \to \infty} \frac{\pi_i(t+h) - \pi_i(t)}{h} = \sum_{j \in \text{states}} q_{j,i} \pi_j(t) \]
which, also has the form

$$\pi'(t) = \pi(t)Q$$

To solve the steady state probabilities, we only need to solve

$$\pi Q = 0, \quad \sum_{i \in \text{states}} \pi_i = 1$$

More detail on the numerical analysis will be covered in section 2.2.

### 2.3.3 Semi-Markov Processes (SMP)

In the CTMC, the Markov process had the property that a transition was made at every time instant. That transition may well return the process to the same state, but a transition occurred nevertheless. In the SMP, the time between transitions may be several unit time intervals, and this time may depend on the particular transition being made. The process is no longer strictly Markovian, thus it is known as a semi-Markov process [6].

The SMP can be thought of as a process whose successive state occupancies are governed by the transition probabilities of a Markov process, but which spends time in any state described by an integer-valued random variable that depends on the state currently occupied and on the state to which the next transition will be made. At the transition instants, the semi-Markov process behaves just like a Markov process. This is called the embedded Markov process. This embedded Markov process can be analysed relatively easily.
2.3. Markov Chains

Note that there is only one random variable that depends on the state currently occupied. This characteristic (or restriction) will be further discussed in Section 2.4.4.

2.3.4 Generalised semi-Markov Processes (GSMP)

Sojourn times in CTMCs are restricted to the negative exponential distribution, and therefore CTMCs have limited expressiveness. GSMPs, however, do not have that restriction. They are generalisations of CTMCs, and they can have actions that are governed by general distributions. Then, strictly speaking, GSMPs are not Markov chains because they lack the memoryless property.

Hillston and Pooley [45] describe a GSMP as a process in which each state is characterised as a set of active elements, each of which has an associated lifetime. A state change occurs when an active element completes a lifetime and all residual interrupted elements record their lifetimes. Whenever the element is active again it resumes its remaining lifetime. Of course, if the lifetime is exponential we may disregard the residual lifetime.

We can see from the description above that GSMPs can describe processes with

- fixed time delayed transitions - eg. exactly 20s
- uniform distributed time delayed transitions
- immediate transitions

and they therefore are very expressive. However, because they do not have the Markovian property, they are almost impossible to solve analytically or numerically. It seems that simulation is the only solution method.

2.3.5 Markov Chains lack compositionality

Markov Chains are flat structures that are difficult to design and understand. Many complicated systems result in Markov chains that are over a few million states in size; it is impossible to derive a Markov chain by hand.

Therefore, Markovian models are typically constructed from some other high-level formalism such as queuing networks, stochastic Petri nets, stochastic activity networks, or stochastic process algebras, some of which were described in the previous sections.
2.4 Petri nets

Petri nets are abstract formal methods, invented in 1962 by Carl Adam Petri [6], for the description and analysis of flow of information and control in concurrent systems. Petri nets are graphically represented as collections of

**places** which are represented by circles. Places model conditions, or state variables. Places can contain **tokens**.

**tokens** are represented as black dots. Tokens can only appear in **places**. Tokens represent the specific value of the condition, or of the state variables.

**transitions** are drawn as rectangles. They model activities which cause a change in state.

**arcs** are arrows between **places** and the **transitions**. They specify the interconnection between the two types of objects. Note the Petri nets are bipartite graphs. That is, only places are connected to transitions and vice versa, but no two places or transitions are connected.

The graphical aspect of these models are attractive for practical modelling since they help in understanding how features of the real system are conveyed in the model. An example of a simple Petri net is shown in Fig. 2.3.

![Figure 2.3: A simple Petri Net](image_url)

Classic PNs are independent of time and environment, and they are characterised by the non-deterministic firing of transitions that are simultaneously enabled in a given marking. A marking is an assignment of tokens to places. The marking of a Petri net determines the state of the Petri net. Therefore, a marked Petri net can be defined formally as a 5-tuple:

\[
PN = (P, T, I^-, I^+, m_0)
\]

where \( P = (p_1, p_2, \ldots, p_P) \) is the set of places, \( T = (t_1, t_2, \ldots, t_T) \) is the set of transitions, \( I^-, I^+: (P \times T) \rightarrow \mathbb{N} \) are the backward and forward incidence functions, and \( m_0 = (m_{01}, m_{02}, \ldots, m_{0P}) \) is the initial marking.

Transitions can be enabled, which means that they can fire. Firing means that the transition removes from its input places a number of tokens, defined by the weight of the input arc. It also
adds to its output places the number of tokens defined by the weight of the output arc. In this report, grey boxes represent enabled transitions, whereas empty boxes represent transitions that are not enabled.

Fig. 2.4 shows an animated example of enabling and firing. Filled (black) rectangles represent enabled transitions, whereas the white rectangles represent disabled transitions. Only enabled transitions are allowed to fire. When a transition fires, one token is removed from its input place, and a token is placed at its output place. Fig. 2.5 shows a slightly more complicated example,

![Figure 2.4: An animated example](image1)

with two tokens. We can derive the *Reachability Graph* of the Petri net by determining all the markings which result from the different transitions firing. For example, the initial marking (the leftmost net in the diagram) is \((2,0,0)\). After transition \(t_1\) fires, the marking is now \((1,1,0)\). Now, two transitions are enabled. If transition \(t_2\) fires, the marking will become \((1,0,1)\). The corresponding Reachability Graph of this example is shown in Fig. 2.6.

![Figure 2.5: A more complicated example](image2)

### 2.4.1 Properties of Petri nets

As a modelling formalism, PNs do have many properties formally defined mathematically. We shall not go into these *gory* details; instead, I will present to you examples of Petri nets
demonstrating these properties visually. For details, please consult Balbo [5].

**Reversibility**

A PN is reversible if and only if from any state reachable from the initial marking $m_0$, it is possible to come back to $m_0$. This property is shown in Fig. 2.7

**Liveness**

A transition is *live* if and only if, for each marking $m$ reachable from $m_0$, there exists a marking $m'$, reachable from $m$, such that the transition is enabled in marking $m'$. A PN is live if and only if all transitions in the net are live. A transition that is not live is *dead*. If at least one transition is live, then the PN cannot *deadlock*. If all transitions are live, then the PN contains no *livelock*. Fig. 2.8 shows an example of a Petri net which is live and another which is not.

---

2. a condition when it enters a state in which no transition can be fired
3. a condition when it enters a subset of its activities from which it has no possibility of exiting
2.4. Petri nets

Boundedness

A place $p$ of a PN is $k$-bounded if and only if for each reachable marking $m$, the number of tokens in that place is less than or equal to $k$. A Petri net is $k$ bounded if and only if all places in the Petri net are $k$-bounded. PN that are 1-bounded are said to be safe. Boundedness implies the finiteness of the state space. Fig. 2.9 gives an example of a bounded and an unbounded Petri net.

2.4.2 Stochastic Petri nets

Stochastic PNs are timed PNs in which all the firing delays are exponentially distributed. The use of these exponential distributions for the temporal specifications results in a PN that can be mapped on continuous-time Markov chains, which are discussed in section 2.3.

SPNs are formally defined by adding the set $\Lambda = (\lambda_1, \ldots, \lambda_2)$ to the definition of a Petri net. $\lambda_i$ is the transition rate of the transition $t_i$. As the transition delays are exponential, this $\lambda$ is also the parameter for the exponential distribution.

If two transitions are enabled at the same time, the transition that fires first will have the minimum delay. This is also known as the race condition. If $t_1$ and $t_2$ are both enabled, the probability that $t_1$ will fire first is given by $\frac{\lambda_1}{\lambda_1 + \lambda_2}$. 
The quantitative analysis of SPNs can be carried out by analysing the underlying Markovian process. This Markov process is essentially the same as the Reachability Graph, as described in page 17 and shown in Fig. 2.6.

2.4.3 Generalised Stochastic Petri nets

GSPNs\(^4\) are SPNs with the introduction of immediate delays. There are now two types of transitions, *immediate* and *timed*. Immediate transitions fire in zero time once enabled. In most books, immediate transitions are represented as filled black boxes / bars.

The use of immediate transitions results in a more expressive PN but with little change to the analysis of the underlying CTMC. They are analysed by decomposing the markings into subsets: *vanishing* and *tangible*. Vanishing markings are those in which the the system spends zero time in, that is, the markings that involve immediate transitions. Similarly, the markings that involve timed transitions are known as tangible.

With the introduction of immediate transitions, some consideration is needed with regard to the qualitative analysis; not all properties of Petri nets are inherited by GSPNs. In particular, a Petri net which is live may not be a live GSPN. A GSPN which is live, may also not be live when considered as a timeless Petri net. There are several examples of these Petri nets in Bause and Kritzinger [6].

2.4.4 Semi-Markov Stochastic Petri nets

SM-SPN\(^5\) are extensions of GSPNs[32] which support arbitrary marking-dependent holding-time distributions and which generate an underlying semi-Markov process rather than a Markov process (see Section 2.3.3).

The concept of generating a semi-Markov process from a non-Markovian stochastic Petri net is well established: an SMP can only be generated from such a non-Markovian SPN only if transitions with general firing-time distributions (GEN) are constrained to be *exclusive* [34]. An *exclusive* transition is one that, whenever it is enabled, no other transition is enabled.

In general, when the net has more than one timed GEN transition enabled at any state (said to be *conflicting*), then the pertinent question is: *which one of the enabled transitions is going to fire?* Two selection rules are possible [5]: *preselection* where the enabled transition that will

---

\(^4\)Generalised Stochastic Petri Nets

\(^5\)Semi-Markov Stochastic Petri nets [17]
fire is chosen when the marking in entered; or race, where the enabled transition that will fire is the one whose firing delay is the minimum.

Issues of scheduling policies for residual pre-empted transition times also need to be considered. Three main scheduling policies for pre-empted transition times are described in [34]:

**pre-emptive resume (prs)** the original firing time distribution sample is remembered and work done (time elapsed) is conserved for when the transition is next enabled.

**pre-emptive restart identical (pri)** the original distribution sample is remembered but work done is lost and the transition firing delay starts from 0 when it is next enabled.

**pre-emptive restart different (prd)** the original distribution sample is forgotten, work done is discarded and when the transition is next enabled, the transition firing delay is resampled and starts from 0.

SM-SPNs do not try to tackle the issue of concurrently enabled GEN transitions. If more than one GEN transition is enabled, then a probabilistic choice is invoked to determine which will be fired [17]. That is, SM-SPNs follow the preselection rule. SM-SPNs use a prd schedule for pre-empted GEN transitions. This approach is not a solution to the more complex issue of properly concurrently enabled GEN transitions, but it is merely a way of specifying a different type of model - a semi Markov model where GEN transitions are forced to be exclusive. Where there are no concurrently enabled GEN transitions, then proper concurrent and competitive transition behaviour is catered for with prs scheduling for pre-empted transitions.

### 2.4.5 Queueing Petri nets

QPN\(^6\) is an integration of the concept of a queue with a coloured version of a GSPN. This is done by partitioning the set of places into two mutually exclusive subsets, queued places and unqueued places.

Unqueued places function the same way as before; Queued places, on the other hand, contains two components: the queue and a depository for tokens which have completed their service at the queue. Tokens added onto a queued place by any of its input transitions are inserted into the queue (which may or may not be FIFO). Tokens in the queue are not available for transitions. Only after completion of its service, when it is added to the depository, is the token available for transitions.

\(^6\)Queuing Petri Net [7]
According to Bause and Kritzinger [6] and Bause et al. [7], the addition of queues in the modelling formalism simplify the description of systems, as we are no longer forced to use GSPN elements to describe the scheduling strategies. The complexity of the performance analysis, which is determined by the size of the state space, still remains the same as that obtained by modelling the queue with coloured GSPN elements.

### 2.4.6 Pros and Cons of the Petri net formalism

#### Advantages

Petri net theory is mature in both theoretical exploration and tool development (see Section 2.8). Also, the performance bounds are derivable already from PN-structure; and detailed performance analysis can be done by mapping the marked PN onto a Markov chain, which can then be solved, as detailed in Section 2.2.

Petri nets have also been successfully applied in many situations where synchronization has a determining influence, and also in industrial studies.

#### Disadvantages

It is very difficult to specify a complex system in the Petri net formalism. In fact, the complexity for the specification and representation of large systems is exponential compared to linear in case of process algebras [43]. This proves to be the biggest drawback of the Petri net formalism.

### 2.5 Process Algebras

Process Algebras are languages, defined using some grammar, used for specifying system behaviour [2]. There are several types of Process Algebras in the academic community at present, such as CCS\(^7\), CSP\(^8\), ACP\(^9\), LOTOS\(^10\) and FSP. They can be used for qualitative analysis [54], such as determining liveness, safety, boundedness and fairness properties.

As they are very similar, only the syntax of FSP is presented. The process \(P\) in FSP can be described as:

---

\(^7\)Calculus of Communicating Systems [55]  
\(^8\)Communicating Sequential Processes [46]  
\(^9\)Algebra of Communicating Processes [8]  
\(^10\)Language of Temporal Ordering Specifications [29]
\textbf{STOP} is a Nil process. For example, \( P = \text{STOP} \) means that process \( P \) can engage in \textit{no} action.

\( a \rightarrow Q \) is a prefix action. For example, \( P = (a \rightarrow Q) \) means that the process \( P \) can engage in action \( a \) and then behaves as process \( Q \).

\( Q \mid R \) is a choice operator. For example, \( P = (a \rightarrow X \mid b \rightarrow Y) \) means that process \( P \) can either engage in action \( a \) or action \( b \). If it engages in \( a \), it will behave as process \( X \), and if it engages in \( b \), it will behave as process \( Y \).

\( Q \parallel R \) is a composition operation. For example \( \parallel P = (Q \parallel R) \) describes a composition of the processes \( Q \) and \( R \). \( Q \) and \( R \) run in parallel and synchronise on the set of actions that have the same name.

\( Q \setminus L \) is a hiding operation. The actions in the set \( L \) that emanate from the process \( Q \) is rewritten as silent actions that can no longer be used in cooperation with other components. Similarly, there is also a show operator \( \oplus \). For example, \( (Q \oplus L \) hides all the actions in \( Q \) except those in set \( L \).

\( Q/\{\text{newlabel}/\text{oldlabel}\} \) is a renaming operation. The action \( \text{oldlabel} \) is rewritten as \( \text{b} \). This renamed process happens before composition, and can therefore be used to synchronise two actions that have different names. FSP has some syntactic sugar to simplify renaming. For example, it is possible to give a definition like \( Q/\{a_1/b_1, a_2/b_2, \ldots, a_n/b_n\} \). In fact, \( \text{newlabel} \) and \( \text{oldlabel} \) can also be sets of action labels.

FSP also contains various pseudo-programming constructs, such as iteration, error states, and guarding. Although these are arguably not SPA constructs, they are syntactic sugar that simplify the specification of a model. It is possible to construct a model without these constructs; indeed, the LTSA tool itself converts all these constructs into the basic FSP described above. Details about the syntax can be found in \cite{54}.

It is relatively easy to specify a system using process algebras because it has a natural, imperative syntax. Moreover, because of its compositional nature, a system can be described in many simpler components. Each component can be specified and considered as a separate entity. Specifying systems in this way allows very large and complicated models to be constructed in a clean and structured manner.

This compositional approach also allows tools to minimise on the memory usage. For a model with \( k \) components, each with \( n_i \) states, instead of representing the entire state space with
\[ \prod_{i=1}^{k} n_i \] states, the model can be represented as a composition with a total of \[ \sum_{i=1}^{k} n_i \] states. The tool will still have to traverse the entire state space, but they need not be stored in memory [2]. We will look at the LTSA tool for the FSP tool in Section 2.9.

As Process Algebras do not express random variables, probability and time delays, however, they cannot be used for qualitative performance analysis. In the following section, we will look at SPAs\textsuperscript{11}, which can address this problem.

### 2.6 Stochastic Process Algebras

Stochastic Process Algebras, according to Hillston and Pooley [45], were first proposed in 1990, as a tool for performance and dependability modelling. They are basically stochastic extensions of process algebra [20, 2] and therefore, they can be used for qualitative performance modelling.

Stochastic Process Algebras offer several attractive features over other performance modelling paradigms [13]. The most important of these is compositionality, the ability to model a system as the interaction of its subsystems. Stochastic Process Algebras also offer formal semantics and abstraction, the ability to disregard detailed but irrelevant internal behaviour. Queuing networks offer compositionality but not formality and abstraction, whereas SPN\textsuperscript{12} offer formality but not compositionality and abstraction [45].

The ability to build models iteratively using compositionality allows very complicated models to be specified, but this inevitably leads to models which are extremely large. There have been some research [18] over the last few years on reducing the complexity of the underlying Markov model, and some of the later research [62, 57, 44] describes methods which involve taking advantage of the compositional structure within SPA models.

There are generally two categories of SPAs: Markovian SPAs and Non-Markovian SPAs.

#### 2.6.1 Markovian SPAs

Markovian SPAs are SPAs in which all delays are exponentially distributed. Markov chains are described in Section 2.3.

\textsuperscript{11}Stochastic Process Algebras
\textsuperscript{12}Stochastic Petri Net [6]
PEPA

PEPA was developed in the University of Edinburgh, and is described fully in Clark [25] and Hillston [44]. Like all SPAs, it offers compositionality. It is also described as being parsimonious, i.e., a component is easy to understand.

It has formal semantics, and it is purely Markovian. This results in an SPA that is very much simpler to analyse mathematically than many other SPAs. It has a variety of tools, including the PEPA workbench [35] that translates PEPA into a Maple input language, IPC [14] that converts PEPA into a Petri net description language that DNAma can solve and PRISM [53]. These, and some others, will be described in Section 2.8.

The syntax of PEPA is very similar to other process algebras, and shall be described here. The process $P$ can be described in PEPA as:

$$P ::= (a, \lambda).P \mid P + P \mid P \triangleright_s P \mid P/L \mid A$$

where

$(a, \lambda).P$ is a prefix operation. It represents a process which does an action, $a$, and then becomes a new process, $P$. $\lambda$ is the parameter of the exponential distribution that describes the delay of the action $a$. The parameter may also take a $\top$ value, which makes the action $passive$ in a cooperation (see cooperation below). This prefix operation is much the same as $P = (a \rightarrow Q)$ in FSP, except that now there is a notion of time.

$P_1 + P_2$ is a choice operation. If $P_1$ occurs first then the process behaves as $P_1$ and discards any behaviour of $P_2$, and vice-versa. That is, this is a $race$ condition. This works much the same as $P = Q \mid R$ in FSP, where process $P$ can behave either as $Q$ or as $R$.

$P_1 \triangleright_s P_2$ is a cooperation operator. This means that $P_1$ and $P_2$ runs in parallel and synchronize over the set of actions in set $S$. That is, if $P_1$ reaches a point where it can only evolve with an action $a \in S$, then it must wait until $P_2$ reaches a point where it is also able to produce action $a$. In an $active$ cooperation, where the rates are not $\top$, the two components then jointly produce a synchronised action with a rate that is the slower of the two components. In a passive cooperation, where a component has an action of rate $\top$, then the joint action has a rate of the slower component.

$P/L$ is a hiding operator. The actions in the set $L$ that emanate from the process $P$ is rewritten as silent $\tau$ actions that can no longer be used in cooperation with other components.

$A$ is a constant label. This allows recursive definitions and simplification of the description.
TIPP

TIPP [37] was developed in the University of Erlangen. It is very similar to PEPA, with a few minor differences that is described by Hillston and Pooley [45].

Unlike PEPA, the rate of shared activity is the product of the rate of the synchronising activities (and not the slower). Therefore, 1 is chosen as the rate of passive actions, since any number multiplied by 1 is the same number. Between two active actions, it is assumed that one of the rates represents a *scaling factor* capturing the dimensioning of the required task with respect to some standard measure. Between one active and one passive, the rate of shared action reflects the rate of the active action.

TIPP is also purely Markovian, and being so, have many tools that allow efficient quantitative and qualitative analysis.

Interactive Markov Chains

IMC\textsuperscript{13}, like all SPAs, offers compositional specification and analysis. It differs from most of them, though, by having delays as orthogonal entities, separate from the actions. As Hermanns [41] describes, doing so avoids some of their shortcomings in the definition of synchronisation.

The advantage of separating the delays from the actions is obvious: it allows the analyst more control over the design of the model. In particular, it avoids the need to delay *each* action, unlike PEPA and TIPP.

However, setting the clock and testing the clock is the same action. If we further allow the separation of setting and testing, the expressiveness of the language can be further improved. This is the approach taken in SFSP, as will be described later.

2.6.2 Non-Markovian SPAs

Non-Markovian SPAs are SPAs in which the constraint that all the delays are exponentially distributed are relaxed.

EMPA

Developed in University of Bologna, it was originally called MPA [11], before being improved upon and called EMPA [10]. EMPA is also similar to PEPA, with a few differences.

\textsuperscript{13}Interactive Markov Chains [41]
First, rather not intuitively, the rate of shared activity is the maximum of the rates of the participating activities [20]. Therefore, passive activities are given the rate 0 (since it will be definitely be the minimum of all rates).

Second, EMPA also permits immediate actions, and these can have associated priority levels and associated weights. The addition of this means that the underlying model of EMPA is a GSMP\(^{14}\), rather than a CTMC (discussed further in Section 2.3).

**SPADES\(^{15}\)**

SPADES was developed in Imperial College [39]. However, unlike PEPA or EMPA, it allows delay transitions that come from a general distribution. This means that the delay can be uniformly distributed, normally distributed, or even fixed.

In addition, it has a probabilistic choice construct which allows mapping from queueing networks to SPADES directly. This is not possible in PEPA [39].

♠

♠ (not to be confused with SPADES) is modelled after stochastic automata (inspired by Timed automata), which is based on the notion of clocks. ♠ has the signature of a Process Algebra, but with clock guarding and setting. This approach is also taken in SFSP.

♠ can be directly mapped to stochastic automata, and qualitative analysis can be done through this automata. Being entirely non-Markovian though, it is difficult to perform numerical analysis on the model, and therefore it uses discrete event simulation to obtain performance results.

**MoDeST [28]**

MoDeST\(^{16}\), according to Ayles [2] is not strictly a stochastic process algebra. It is a modelling language, that combines conventional programming constructs such as iteration, alternatives, atomic statements and exception handling.

It also incorporates means to describe non-determinism, probabilistic branching, and various stochastic aspects [28]. It is very similar to SFSP, as SFSP has also the same constructs.

\(^{14}\text{Generalised Stochastic Markov Process [6]}\)

\(^{15}\text{Stochastic Process Algebra for Discrete Event Simulation [39]}\)

\(^{16}\text{Modelling and Description Language for Stochastic Timed systems [28]}\)
2.6.3 SFSP

SFSP is an extension to FSP, which was described earlier in Section 2.5. It was extended with the aim to adopt the stochastic timed automata as its semantic basis. Therefore, it is grounded in terms of formality and is more expressive than many other stochastic process algebras. More information on SFSP can be found in Ayles [2].

SFSP allows the specification of probabilistic choice, which, in Markovian SPAs, were simulated as the rate of the action. Like SPADES, it allows delay transitions that come from a general distribution. It also separates the action of clock setting and guarding from the actions.

The stochastic extensions of FSP are described here [3]:

**Probabilistic Choice**  
\[ P = ( (a) \ Q \ | \ (b) \ R ) \]
This represents a process \( P \) that behaves as \( Q \) with probability \( \frac{a}{a+b} \), or as \( R \) with probability \( \frac{b}{a+b} \). For example, if \( a \) is 2 and \( b \) is 1, then \( Q \) will occur twice as often as \( R \).

**Clock Setting**  
\[ P = ( <c:\text{exp}(\text{RATE})> a \rightarrow Q ) \]
means that just before the action \( a \) occurs, the clock \( c \) is set to be an exponentially distributed random variable with parameter \( \text{RATE} \). Clocks can take values from any general distribution. There are several already inbuilt in the LTSA tool, such as Uniform, Fixed, Erlang, Gamma, Geometric and the Normal Distribution.

**Clock Testing**  
\[ P = ( ?c? a \rightarrow Q ) \]
means that the clock is tested for zero before action \( a \) can occur. Therefore something like \( P = ( <c:\text{fixed}(5)> ?c? a \rightarrow Q ) \) means that action \( a \) can only occur exactly after 5 time units. It is possible to test the clock several actions after the clock is set. In fact, SFSP even allows setting and testing several clocks before an action.

In addition, SFSP allows the specification of performance measures together with the model. It is similar to stochastic probes (see Section 2.7). There are three types of performance measures that can be specified in SFSP:

**Event Counter**  
This records the number of occurrences of certain events during the simulation. For example, \( \text{counter } E = \{ a, b, c \} \) counts the number of each of \( a, b, \) and \( c \) that occurred during the simulation.

**Population Counter**  
This is usually used in combination with a queue. It requires specifying at least two events, one to decrement the counter and one to increment it. For example, when used with a queue with two actions \( \text{enqueue} \) and \( \text{dequeue} \), the code
2.7. Specifying performance measures

The specification of performance measures in the modelling formalisms described above is not a trivial issue. In this section we will look at two ways that we can specify performance measures, and compare the two. First of all, let us look at the types of performance measures that we would like to specify.

2.7.1 Types of performance measures

There are different ways of classifying performance measures. One way of classifying performance measures is described by Herzog [43] as:

**User View** These concerns measures to do with the customer, or the user, or the object in the system. Measures such as waiting time, response (sojourn) time and total delay all fall under user view. Note that we may not just be interested in the mean values, but also the variance, skewness, or even the distribution function of these measures.

**System View** These concerns measures to do with the system. For example, this category includes the utilization and throughput of different resources, number of waiting customers at various locations, probability of overflow in which the buffer size or queue size is insufficient etc..

Argent-Katwala et al. [1] describe another categorisation of performance measures:

```plaintext
measure N <enqueue, dequeue>
}
```

specifies a counter that can measure the mean population over time.

**Timer** This is usually used to determine the passage time of a system. For example, \( \text{timer } T < a \rightarrow b > \) specifies the mean passage time between state \( a \) of a system to state \( b \). This measure is implemented by using a FIFO queue, to store the current time whenever an \( a \) action occurs, and to subtract from the current time the first time in the queue whenever \( a \rightarrow b \) action occurs. This subtracted time is then averaged to find the mean passage time in a system.
Steady State measure This concerns the state of the system when it reaches equilibrium (if it does). Typically these are the results that analysts are most interested in. Most of the measures as described above in user view and system view can be steady state measures as well. For example, utilization, throughput, queue length are usually steady state performance measures.

Transient State measure This concerns the state of a system at a definite time instance. For example, we may be interested in knowing the probability that a just-in-time compiler is running native code exactly 5 seconds after loading a Java applet [1]. Or we may be interested in knowing the probability that a failed server takes 3 seconds or less to recover.

Passage Time measure This concerns the time it takes for a system to evolve from one set of states to another set of states. For example, we may be concerned with the time it takes for a web server system to receive a read request and send a reply. Or we may want to find out the time taken to process some document.

2.7.2 Logic based approach

Formal logics for asking performance related questions of stochastic systems provide a concise and rigorous way to pose such questions, and even allow for the composition of simple questions into more complex queries [17]. Moreover, the study of temporal and modal logics in conjunction with models of concurrency is well established [24]. These logics express properties of systems which have a number of states and in which there is a relation of succession. A modal logic is used to express a finite behaviour, while a temporal logic is used to allow reasoning over infinite behaviour.

One such logic is CSL [4]. CSL can express performance measures by selecting states and paths from a system that meets both steady-state and passage time quantile criteria. Bradley et al. [17] have developed an extended version of CSL (eCSL) which augments CSL with the ability to express a richer class of passage time quantities as well as measures based on transient distributions.

Clark et al. [24] have developed a performance specification language based on the formal foundations of stochastic logic. However, the theory behind specifying transient measures is not well established as yet.

Here, we present the syntax of eCSL and several examples that are described in Bradley et al. [17]. An eCSL statement, \( \Psi \), is defined by

\[
\Psi \overset{def}{=} tt \mid \Psi \land \Psi \mid \neg \Psi \mid S_{p}(\psi) \mid T_{p}^{\tau}(\psi, \psi) \mid \sigma_{p}^{\tau}(\psi, \psi)
\]
\[ \psi \overset{\text{def}}{=} \mathsf{tt} \mid p[N] \mid \psi \land \psi \mid \neg \psi \]

If a single state \( s \) satisfies each of the possible clauses of a \( \Psi \) formula, it is written \( s \models \Psi \). Some of the formal semantics of eCSL are:

\[
\begin{align*}
    s \models \mathsf{tt} & \quad \text{for all } s \\
    s \models \Psi_1 \land \Psi_2 & \quad \text{iff } s \models \Psi_1 \land s \models \Psi_2 \\
    s \models \neg \Psi & \quad \text{iff } s \not\models \Psi
\end{align*}
\]

\( \psi \) is a state specification formula, separate from the \( \Psi \) performance specification formula, unlike in the original CSL. \( \text{Sat}(\psi) \) is defined as \( \text{Sat}(\psi) = \{ m \in M \mid m \models \psi \} \).

\( N \) is defined as \( N \in 2^N \) and \( p[N] \) is satisfied is the number of tokens on place \( p \) in some state \( m \) is in the set of allowed numbers of tokens \( N \). \( \rho \in 2^{[0,1]} \) is a set of allowed probabilities and \( \tau \in 2^{[0,\infty]} \) is a set of times.

\( \text{Sat}_\rho(\psi) \) is true if the steady state probability of being in the set of states defined by \( \psi \) lies in the set \( \rho \). \( \text{Sat}_\rho^\tau(\psi, \psi) \) is satisfied by a set of start states if the probability of the system being in states \( \text{Sat}(\psi_1) \) at time \( t \), while not having passed through states \( \text{Sat}(\psi_2) \), lies in \( \rho \) for all times \( t \in \tau \). Finally \( \text{Sat}_\rho^\tau(\psi, \psi) \) is true for a set of start states if the random variable representing the passage time to target states \( \text{Sat}(\psi_1) \), while not having traversed states in \( \text{Sat}(\psi_2) \), lies in the range of times \( \tau \) with probability \( p \in \rho \).

Here are some examples of eCSL:

\[
\text{Sat}(p_1[35] \land p_5[10], p_2[175], p_6[1]) \models \text{P}_{(0.10)}(p_2[175], p_6[1])
\]

This asks the question about the probability that the passage time, defined by the source states \( p_1[35] \land p_5[10] \), the target states \( p_2[175] \) and the excluded states \( p_6[1] \), takes less than time 10. Note that these \( ps \) define sets of states, for example, \( p_1[35] \land p_5[10] \) selects all the Petri net markings which have 35 tokens in \( p_1 \) and 10 tokens in \( p_5 \).

\[
\text{Sat}(p_1[35] \land p_5[10], p_2[175], p_6[1]) \models \left( \text{P}_{(0.9,1)}(p_2[175], p_6[1]) \land \text{P}_{(0.98,1)}(p_2[320], p_6[4]) \right)
\]

This expresses a requirement that we need to achieve a 90% quantile for a passage time within the first 10 time units of the passage starting, and a 98% quantile over a different passage, within the first 100 time units.

We can immediately see that a major drawback of the logic based approach is that it is notationally more complex [1] and may be seen as esoteric [24], requiring more effort by the user.
to understand and apply.

### 2.7.3 Process algebra based approach

There are ways to specify performance measures in the process algebra of the model. Bernado [9] extended the syntax of the stochastic process algebra EMPA such that activities include a notion of reward. This can be used to generate a reward structure; the reward assigned to each state is the sum of the rewards associated with the activities the model enables in that state. By assigning different values to the reward field, the user is able to calculate performance measures such as utilisation, throughput and system uptime.

Ayles [2] also extended the syntax of SFSP such that simple performance measures, such as the count of an event, the population of a queue, and passage times between two sets of states can be specified in the syntax. These performance counters are composed like any other process together with the system. They keep track of when a particular action occurs, and then updates the appropriate measure when necessary.

Argent-Katwala et al. [1] has also come up with a novel way of specifying performance measures in process algebras by using what they call Stochastic Probes. These probes, like the method used by Ayles, are composed together with the system, and keep track of when a particular action occurs.

We will present the method as described by Argent-Katwala et al. [1] here. A stochastic probe has the following syntax:

\[
R ::= a | T, T | (S)
\]

\[
S ::= T | T|S
\]

\[
T ::= R | R^n | R^m | R^d | R^e | R^f
\]

\[
a ::= act | act : start | act : stop
\]

To understand the examples of the stochastic probes presented by Argent-Katwala et al. [1], it is necessary to reproduce a simple PEPA model. PEPA was chosen to demonstrate the stochastic probes because of its simple syntax.

\[
Light_1 \overset{def}{=} (red, r_1).(green, r_2).(blue, r_3).Light + (fail, r_4).Light_2\]

\[
Light_2 \overset{def}{=} (repair, r_1).Light_1
\]
This model describes a light that oscillates between red, green and blue, and can periodically fail, and then repair itself.

Here are some examples:

\[(\text{red : start, fail : stop})\]

this describes a probe that measures the time to failure. It does this by waiting for a red action, then absorbs any other behaviour before changing state again on seeing a fail.

\[((\text{fail, red : start, red : stop} | (\text{red : start, red : stop}))\]

is a probe which can analyse the time between successive red actions after a fail or in normal operation.

\[(\text{green : start, blue}^3, \text{blue : stop})\]

is a probe that looks for the occurrence of a green action followed by four blue actions.

We can see that a major advantage of the process algebra based approach is that it is relatively simple to use and apply, and reasonably expressive. It does not require the user, who is already familiar with using the algebra to specify the model, to learn another language for the specification of the performance measures.

### 2.8 Survey of existing tools

#### 2.8.1 SPA tools

**PEPA Workbench [35]**

The PEPA Workbench does checking of the well-formedness of a PEPA model through the creation of the state transition diagrams and also does the calculation of performance measures based on the infinitesimal generator matrix [23]. It does not do the calculation directly; rather, it converts this matrix into an interface to the Maple Symbolic Algebra package for the solution of the matrix equations.

**Imperial PEPA Compiler (ipc) [14]**

ipc is a tool that can process PEPA-specified passage-time densities and models by compiling the PEPA model and passage specification into the DNAmaca formalism. This formalism is based on the Petri-net formalism.
**PRISM [53]**

PRISM\textsuperscript{17} is a probabilistic symbolic model checker. It stores the generator matrix as a multi-terminal binary decision diagram (MTBDD) which offers compact storage for structured state spaces of significant size. It offers its own PRISM input language, which is very similar to a stochastic process algebra. In addition to that, it also supports PEPA as one of its input languages. It offers a range of numerical solution procedures, for example, Power, Jacobi, Gauss-Seidel, JOR, and SOR.

**TIPPtool [37, 43]**

TIPP tool, developed by Götz et al. [37], has a LOTOS oriented input language. It is able to analyse a system's functional behaviour by reachability analysis and temporal properties by numerical methods. There are various numerical solution modules available for transient and stationary analysis of CTMCs.

The tool is also able to do exact or approximate compositional reduction based on user guided exploitation of symmetries. According to Herzog [43], 100k states solved easily with the TIPP tool.

**2.8.2 Petri-net tools**

**DNAmaca[49]**

DNAmaca, now recently known as HYDRA [15], is a complete performance analysis tool, which includes the model specification language, state space generation, functional analysis, and the computation of steady state solutions. It has been recently extended to allow even the derivation of transient states and passage time distributions [31].

The possible steady state solutions offered by DNAmaca include direct methods (Gaussian Elimination, Grassmann), classical iterative methods (Gauss-Seidel, fixed SOR, dynamic SOR), Krylov subspace techniques (BiCG, CGNR, CGS, BiCGSTAB, TGQMR) and decomposition based methods (Aggregation-Isolation, Aggregation-Isolation Relaxed) [14].

The HYDRA release [15] of DNAmaca can calculate passage-time quantiles in Markov models, by using a uniformisation technique. This is described a little in Bradley et al. [14].

\textsuperscript{17}Probabilistic Symbolic Model Checker [53]
2.8. Survey of existing tools

**PIPE**

PIPE, or Platform Independent Petri net Editor \(^{18}\) is a Petri net editor/ animator that accepts PNML (Petri net Markup Language) as input. It has several analysis modules, which can perform tasks such as invariant analysis, state space analysis, simulation, and GSPN analysis. Petri nets can be also be exported as PNG and Postscript files for printing or inclusion with other programs or documents.

**QPN-Tool [7]**

QPN-Tool supports the specification and analysis of hierarchically combined Queuing Petri nets (HQPNs). HQPNs are an extension of QPNs allowing the refinement of places by QPN subnets and queues. QPN-Tool is able to analyse both the qualitative and the quantitative aspects of the model.

The quantitative analysis can be performed using the conventional analysis techniques on the flat Markov chain, which is feasible for small state spaces of about 50,000 states, or by the hierarchical technique, which takes advantage of the hierarchical structure of the HQPN, which allows the analysis of models with several millions of states on standard workstations [7]. The tool can even perform approximate analysis based on aggregation.

2.8.3 Queuing network tools

**MACOM [58]**

MACOM\(^{19}\) is a software tool for the model based performance evaluation of communication systems. Models are specified by graphical interactive means, and the solution is derived by using numerical techniques for Markovian models. It is based on USENUM, developed in 1987 by the same author.

MACOM includes several solution techniques, such as direct (Grassman algorithm) and iterative (JOR, SOR). It can also perform aggregation/disaggregation steps to speed up the analysis. According to Sczitnich and Muller-Clostermann [58], it can solve systems with up to 500k states.

\(^{18}\)http://petri-net.sourceforge.net/

\(^{19}\)tool for Markovian Analysis of COmmunication systems [58]
**XMARCA [60, 59, 61]**

XMARCA is a tool for the graphical specification of queueing networks, which can then be solved by numerical procedures. A large selection of solution methods are made available, for example, for steady state solutions, one can use Gaussian Elimination, SOR, SSOR, together with iterative aggregation/disaggregation. There are also Randomization, Runge-Kutta, and other solution methods for transient solutions.

### 2.8.4 Multiple formalisms

**Möbius [26]**

Möbius is a tool for building performance and dependability models of stochastic, discrete-event systems. It supports many different model solution methods and model specification in multiple modelling formalisms. Clark and Sanders [26] developed this tool because they believed that no formalism has shown itself to be the best for building and solving models across many different application domains.

Möbius defines a formal, mathematical modelling framework, with which other formalisms are integrated. It also provides an abstract functional interface that facilitates inter-model communication as well as communication between models and solvers. Clark and Sanders [26] has developed a mapping from PEPA to the abstract functional interface, thereby giving Möbius users the opportunity to use PEPA models in their performance and dependability models.

### 2.9 LTSA and SFSP

In this chapter, we have investigated the various modelling formalisms. In particular, we have looked at FSP in Section 2.5 and SFSP in Section 2.6, and discussed the advantages of performance modelling in SFSP. Here is a summary of the advantages of SFSP over the other formalisms:

- As an SPA, SFSP provides a hierarchical, compositional way to build models. It is also formal in semantics.

- As a non-Markovian SPA, SFSP does not have the restriction that all delays must be exponentially distributed. This, of course, may result in a model where numerical analysis is not possible.
The LTSA tool also has several advantages of the other tools, such as:

- Portability because it was implemented in Java
- Ability to do qualitative analysis
- Ability to do quantitative analysis by simulation
- Extensibility of the tool by having a plug-in interface, due to recent work by Chatley [22].
Chapter 3

Translating SFSP to a Petri net

As discussed in the previous chapter, stochastic Petri nets are well understood formalisms that have many Markov chain analysers, DNAmaca being one of them. DNAmaca can generate and process both Markov and semi-Markov processes [49] using semi-Markov stochastic Petri nets (SM-SPN) as input. By translating SFSP into a SM-SPN, we can directly (numerically) derive performance measures from the model.

Note that SFSP allows a description of programs which do not conform to Markov and semi-Markov processes. This is discussed further in Section 3.8.

Because it is so complex, we present the semantics, the equivalence of SFSP to a Petri net, and the translation scheme in several steps:

1. We first consider SFSP without probability and other features such as alphabet extensions, hiding, relabling and priority. For this subset, SFSP−, we present the following:
   - The semantics of SFSP− (Section 3.3). First, a mapping from SFSP− to stochastic automata is given, then the operational semantics of SA is presented.
   - The equivalence of it and Petri nets (Section 3.4). We construct a corresponding Petri net from some SFSP− code and explore its equivalence.
   - A syntactic translation scheme to convert SFSP− into a Petri net (Section 3.5).

2. Hiding, relabling, alphabet extensions, and priority will then be dealt with separately. For each of these, a mapping from SFSP to the underlying stochastic automata is given, the semantic rules of the underlying stochastic automata is presented, and we show how it can be converted into a Petri net.
3. Performance measures will be discussed in Section 3.7. An informal semantics of the performance measures are given, and how we can convert performance measures in SFSP to performance measures in a Petri net.

4. Probability will be discussed in Chapter 4.

### 3.1 Grammar of SFSP

To simplify matters, we consider a stripped-down version of SFSP. This stripped-down version is almost the same as basic FSP, described in [54], but with the addition of clocks. Basic FSP is FSP without the convenient pseudo-programming structures and syntactic sugar. Note that any FSP specification can be constructed just using basic FSP. In addition, this stripped down version also leaves out some syntax structures regarding composition, alphabet extensions, relabelling, hiding, priority, performance measures and probability. The syntax for composition is left out because we can assume that all the process definitions defined in the language will be composed together - otherwise they can be safely left out of the definition. Alphabet extensions, relabelling, hiding and priority are dealt with in Section 3.6 while performance measures will be discussed in Section 3.7. The issue with probability is described in Chapter 4.

The grammar of SFSP is presented here:

\[
\begin{align*}
FSP & ::= D_1 \ldots D_n \\
D & ::= I = P \,, \, I = P , \, I_1 = P_1 , \, \ldots , \, I_n = P_n \\
P & ::= \text{STOP} \, | \, (\text{Choice}) \\
\text{Choice} & ::= \text{Prefix}_1 \rightarrow P_1 \, | \, \ldots \, | \, \text{Prefix}_n \rightarrow P_n \\
\text{Prefix} & ::= A \, | \, ? \, \text{Test} \, ? \, A \, | \, A < \text{CActs} \\
\text{Test} & ::= C_1 , \, \ldots , \, C_n \\
\text{CActs} & ::= \text{CAct}_1 , \, \ldots , \, \text{CAct}_n \\
\text{CAct} & ::= C : D \, | \, C : \text{hold} \, | \, C : \text{resume}
\end{align*}
\]

where \( I \) refers to the identifiers of processes, \( A \) refers to an action name, \( C \) refers to a clock name (which is guarded), \( P \) is a local process definition and \( D \) is a single process definition. Note that we assume that the process definitions \( D_1 \ldots D_n \) are composed together\(^1\). We can also safely assume that all the local definitions of a single process definition reference each other, otherwise they can also be left out.

\(^1\)If they are not, then we can simply omit them
It is important to emphasise that *clock actions* are different from *clock conditions*. Clock actions \((CAct)\) are clock sets, clock holds and clock resumes, whereas *clock conditions* \((C)\) is a clock test.

### 3.2 Informal semantics of SFSP

Ayles [2] has briefly touched upon the semantics of SFSP in his report. While it does a fair job of formalising SFSP, it has to be elaborated upon, especially when there is no mention of clock holding and clock resume actions in his report.

In SFSP, all actions are considered *urgent*, i.e., an action is performed as soon as it is able to be performed. This is required for a simulation to progress and is often the case for many other process algebras and simulators. Therefore, in a given state, a system will transit out of that state as soon as any transition is enabled.

The only difference between SFSP and FSP is the addition of clocks. The semantics of FSP is described in detail in [54]. Clocks will be elaborated below. The other features of SFSP, such as alphabet extensions, hiding, relabling, priority and performance measures will be discussed later on in Sections 3.6 and 3.7. Probability will be dealt with in Chapter 4.

#### 3.2.1 Clocks

The issue of clock setting and clock testing was dealt with quite thoroughly in [2], but it will still be described here for completeness. An action can have several *clockset actions* associated with it. This means that when an action occurs, the clocks associated with these clockset actions are set. Note that Ayles’ never mentioned that clocks can be reset - though this can be derived from the formalisation he put forward.

An action can also have several *clock conditions* associated with it. These are called *guards* in [2]. An action can only be performed if all the clock conditions associated with it are satisfied (i.e., if the clocks have expired).

For example, for an action prefix \(?gs? \ a \ <cs> \rightarrow \ p\), the guards \(?gs?) must be satisfied before the action is performed. When the action is performed, the clocks \(<cs>\) are set. Note that the guards must be satisfied before the action is performed and the clocks are set.

There remains a question. What does it mean if a clock is tested when it has not been set? In Ayles’ formalisation, each clock action assigns values to the clock variables. For example, \(c: \exp(1)\) assigns \([c = 0, v_c = \Exp(1)]\). If the clock has not been set, then the clock variables
are never initialised, and it is impossible to determine the the behaviour when a guard is translated (in [2], \( c \geq v_c \) yields the condition \( c \geq v_c \). Since both \( c \) and \( v_c \) are not initialised, \( c \geq v_c \) is not sensible).

In the simulation implementation of SFSP, the testing of non-existing clocks does not have an impact on the behaviour of the system. It is as though there never was a clock test. The case where there is a clock test before a clock set is worse: the simulation engine waited for the clock to expire and resulted in LTSA hanging.

The semantics of clocks become more complicated with introduction of clock resume and clock hold actions. Basically, clock hold actions stop a clock, while clock resume actions resume a clock. This meaning is clear until the issue of having clock resume actions without clock hold actions, clock hold actions without clock resume actions, or multiple clock hold/resume actions is raised.

Again, by intuition, we would expect the following:

- Clock holding actions, if the clock is already stopped or it has already expired, does nothing. If the clock is running, then it stops the clock.

- Clock resume actions, if the clock is already running or it has already expired, does nothing. If the clock is stopped, then it resumes the clock.

- If there is a hold action without a resume action, there may be an error. However, it may be the case that the system is still correct and live. The system is still correct, but a warning should be thrown.

- If there is a resume action without a hold action, the system should be expected to behave as though there never was a resume action, since the clock will never be stopped.

Note that multiple clock hold/resume actions are never queued. They either affect the system by stopping or resuming the clock, or they are lost.

### 3.3 Formal semantics of SFSP

While Ayles [2] attempted the formalism of SFSP in terms of stochastic timed automata, we will instead attempt to formalise the semantics of SFSP in terms of stochastic automata. The reasons for choosing stochastic automata over stochastic timed automata are:
3.3. Formal semantics of SFSP

- Ayles did not give a complete mapping from SFSP to STA. In addition, the model of probability is inadequate (discussed in Chapter 4).

- STA is more expressive than necessary. This results in a lot of unnecessary clutter.

- Stochastic automata are well studied formalisms (see [48]) with concrete semantics. It turns out that when we ignore probability, the mapping from SFSP to SA is straightforward.

Stochastic automata is defined to be the tuple \((S, s_0, A, C, \rightarrow, \kappa)\) where

\(S\) is a non-empty set of locations (or states).

\(s_0 \in S\) is the initial location.

\(A\) is a set of actions

\(C\) is a set of clocks

\(\rightarrow \subseteq S \times A \times \mathcal{P}_f(C) \times S\) is a set of edges. \(\mathcal{P}_f(C)\) is a finite set of clocks.

\(\kappa : S \rightarrow \mathcal{P}_f(C)\) is a clock setting function.

Let \((s, a, c, s') \in \rightarrow\) denote \(s \xrightarrow{a,c} s'\). This means that action \(a\) is enabled only after the clocks \(c\) has expired, and after the \(a\) is performed, the stochastic automata moves to state \(s'\).

In order to handle arbitrary distributions, as well as clock stopping and resume, we modify \(\kappa\) to be: \(\kappa : S \rightarrow \mathcal{P}_f(C, D)\), where \(D\) can be any clock distribution, or a clock hold or resume action.

### 3.3.1 Mapping SFSP\(^-\) to SA

Here, \(Q\) ranges over process identifiers, \(E\) ranges over SFSP expressions, and \(M(Q)\) represents the translation of process \(Q\) to a stochastic automata.

**Process definitions**

- \(Q = E\) means that \(M(Q) =_{def} M(E)\)
- \(M(\text{STOP}) = (\{s\}, \{s\}, \{\}, \{\}, \{\}, \{\})\).
Prefix action and clocks

Let $M(E) = (S, s_0, A, C, \rightarrow, \kappa)$. Figure 3.1 shows the stochastic automata of $M(\forall a < c : d > \rightarrow E)$. Notice that we have an extra location $p_c$ where we set a clock $c$ according to the distribution $d$. This place is needed because clock actions are associated with transitions in SFSP, while in SA, clock actions are associated with locations. If we do not add an extra place that sets the clock, then the clock will always be set whenever we move into that particular location, regardless of the transition that occurred. Separating the clock setting with the states simplifies the solution.

![Figure 3.1: Stochastic automata showing simple prefix action with clock setting and testing](image)

Another point to note is that associated with the location $p_c$ is the clock distribution $d$. This allows the clock to be set with any number of arbitrary distributions (instead of associating just one distribution to each clock). Also, $d$ can be a clock hold action or a clock resume action. In order for this to work, a boolean must be associated with each clock to track if it is counting down or not. A clock hold action will set the boolean to true; similarly, a clock resume action will set it to false. The clock will only decrement when the boolean is false. The model of time is exactly like that described in [48].

The mapping is defined as: Let $M(E) = (S, s_0, A, C, \rightarrow, \kappa)$, then

$$M(\forall a < c : d > \rightarrow E) = (S \cup \{p, p_c\}, p, A \cup \{a, \tau\}, C \cup \{c\}, \rightarrow', \kappa')$$

where

$$p, p_c \notin S \text{ (unique places)}$$

$$\rightarrow' = \rightarrow \cup \{(p, a, g, p_c), (p_c, \tau, \{\}, s_0)\}$$

$$\kappa'(p_c) = (c, d), \ \forall s \in S, \kappa'(s) = \kappa(s)$$

Choice

Let $M(E_i) = (S_i, s_i, A_i, C_i, \rightarrow_i, \kappa_i)$, then

$$M(\forall a_1 < c_1 : d_1 > \rightarrow E_1 | \ldots | \forall a_n < c_n : d_n > \rightarrow E_n) = (S, p, A, C, \rightarrow, \kappa)$$
3.3. Formal semantics of SFSP

where

\[ p, p_{c1}, \ldots, p_{cn} \notin \bigcup_{i=1}^{n} S_i \]
\[ S = \bigcup_{i=1}^{n} S_i \cup \{ p, p_{c1}, \ldots, p_{cn} \} \]
\[ A = \bigcup_{i=1}^{n} A_i \cup \{ a_1, \ldots, a_n, \tau \} \]
\[ C = \bigcup_{i=1}^{n} C_i \cup \{ c_1, \ldots, c_n \} \]
\[ \rightarrow = \bigcup_{i=1}^{n} \rightarrow_i \cup \{(p, a_i, g_i, p_{ci}), (p_{ci}, \tau, \{\}), s_i) \mid 1 \leq i \leq n \} \]
\[ \kappa(p_{ci}) = (c_i, d_i), \ \forall s \in S_i, \kappa(s) = \kappa_i(s) \ 1 \leq i \leq n \]

Recursion

We represent the SFSP process defined by the recursive equation \( X = E \) as \( \text{rec}(X=E) \) where \( X \) occurs in \( E \). For example, \( (X = (a \rightarrow X)) \) is \( \text{rec}(X=(a \rightarrow X)) \). Let \( E[X \leftarrow \text{rec}(X=E)] \) denote the SFSP expression that is obtained by substituting \( \text{rec}(X=E) \) for \( X \) in \( E \). The following then applies:

\[
\frac{M(E[X \leftarrow \text{rec}(X=E)]) \xrightarrow{a,c} P}{M(E[\text{rec}(X=E)]) \xrightarrow{a,c} P}
\]

Composition

In SFSP, actions in one process are synchronised over actions of the same name of another process. The mapping \( M(P_1 \ || \ P_2) \) is defined as \( M(P_1) \ || \ M(P_2) \). The composition rules of the underlying stochastic automata is:

Let \( P = (S_1, p_1, A_1, C_1, \rightarrow_1, \kappa_1) \) and \( Q = (S_2, p_2, A_2, C_2, \rightarrow_2, \kappa_2) \), then

\[ P \ || \ Q = (S_1 \times S_2, (p_1, p_2), A_1 \cup A_2, C_1 \cup C_2, \rightarrow, \kappa) \]

where

\( \rightarrow \) is the smallest relation satisfying

\[
\frac{P \xrightarrow{a,c} P'}{P || Q \xrightarrow{a,c} P' || Q}
\]
\[
\frac{Q \xrightarrow{a,c} Q'}{P || Q \xrightarrow{a,c} P || Q'}
\]
\[
\frac{a \notin A_2}{P || Q \xrightarrow{a,c} P' || Q}
\]
\[
\frac{a \notin A_1}{P \xrightarrow{a,c} P', Q \xrightarrow{a,c} Q'}
\]
\[
\frac{P \xrightarrow{a,c} P', Q \xrightarrow{a,c} Q'}{P || Q \xrightarrow{a,c} P' || Q'}
\]

\( \kappa : (S \times S) \rightarrow \mathcal{P}_f(C, D) \) is defined as \( \kappa(s_1, s_2) = \kappa_1(s_1) \cup \kappa_2(s_2) \)
3.4 Equivalence of SFSP$^-$ and Petri nets

The *equivalence* that in this section refers to *behavioural* equivalence. As a labelled transition system is mainly characterised by the actions that can be performed, we will be comparing mainly the transitions that can be fired from a Petri net.

Instead of exploring whether a particular Petri net is equivalent to SFSP, a corresponding Petri net will be created from the FSP or LTS and shown it is behaviourally identical to the FSP definition.

3.4.1 Single process

We can very easily construct a Petri net from FSP (without clocks). Note that an FSP grammar contains no clock actions, no clock conditions, no probabilities, and no measures. The construction of the corresponding Petri net is essentially the same as that described by Bradley et al. [15, 14], who translated PEPA, a timed process algebra, into a Petri net. The simplicity of this is illustrated below.

For a simple one process system (without composition), a Petri net can be easily constructed from the corresponding LTS by assigning each state in the LTS to a Petri net place, and each action in the LTS to a Petri net transition. A token is placed in the place corresponding to state 0 in the LTS. Note that as there are no time delays in FSP, each action is immediate, and corresponds to an immediate transition. Note also that it does not matter if there are local process definitions. Figure 3.2 shows an example of a simple LTS and how it corresponds to a Petri net.

![Figure 3.2: Example of single process conversion](image)

To check if the Petri net is equivalent, we can generate a reachability graph of the net, and check if the reachability graph is the same as the LTS. Alternatively, a more efficient and convenient
3.4. Equivalence of SFSP and Petri nets

Way to check equivalence is to notice that each state in the LTS corresponds to exactly one place in the Petri net, and the state the LTS is currently in is represented by the place the token is in. There will only ever be one active state in the LTS, this implies that there will ever only be one token in the Petri net. This is the case, as notice that each transition has exactly one input and one output arc; it is impossible to remove or add an extra token. That implies that all the markings of the net are just the places of the net, and by inspection, we can then see that the reachability graph obtained from the single token net corresponds exactly to the LTS.

3.4.2 Multiple processes

When processes are composed together, they synchronise on the set of shared alphabets. That is, if an action is shared, then the action can be performed only when all the processes with the shared action are able to perform that action. This means that we can “combine” shared actions together. This is illustrated in Figure 3.3. In the figure, ⊗ represent a composition between Petri-nets.

\[
P = (a\rightarrow b\rightarrow P). \quad Q = (a\rightarrow c\rightarrow Q). \quad ||S = (P \ || \ Q).
\]

**Figure 3.3:** Simple example showing composition of processes

Observe that the behaviour of the Petri net is the same as the LTS. The shared action can only be performed only when both process are able to perform it. There are always two tokens in this net as it is the composition of two process subnets. If we generate the reachability graph of the net, we can check that the Petri net behaves exactly like the composed LTS. Note that unlike the single process subnet, the Petri net has a different structure from the LTS (or reachability graph).

The resulting composed LTS is structurally different from the composed Petri net, even though they are behaviourally similar. Each process must be individually converted into a Petri net.
before they are composed together. This is because composing a LTS reduces the state information, and it is impossible to reconstruct this information from the composed process.

Consider the case when process $P$ has $n$ actions and process $Q$ has $m$ actions. The composition between these two processes will result in $nm$ transitions, as each action $a$ in $P$ synchronises with each action $a$ in $Q$. An example of synchronising with several transitions is illustrated in Figure 3.4. Note that in this example, $P$ has two shared actions while $Q$ has one. The result is $(2 \times 1)$ shared transitions.

![Diagram](image)

$P = (a \rightarrow a \rightarrow b \rightarrow P)$. $Q = (a \rightarrow Q)$. $S = (P || Q)$.

**Figure 3.4:** Example of synchronising several transitions

Again, by inspection, we see that the converted Petri net must behave exactly like the composed LTS.

### 3.4.3 Clocks

It is possible in SFSP to set and test a clock anywhere in the system. In fact, it is even possible to have multiple sets, resets, tests, holds and resumes anywhere in the system.

Because SFSP is so expressive, the construction of a Petri net from SFSP definitions with clocks becomes very complicated. Therefore, it is explained in three parts.

First, only a subset of FSP that precludes locally-defined sub-processes and only allow for clock setting and clock testing will be considered. Processes may call itself recursively, but it cannot refer to other named processes. Second, clock holding and clock resume operations will be added in the grammar. Lastly, we allow processes to have local definitions.
Basic clocks

The idea is to separate clock setting and testing from the rest of the net. Setting a clock places a token in a clockset place, while testing a clock checks a place that contains a token when the clock has expired. Accordingly, setting a clock by a prefix of the form $\langle c:dist \rangle$ builds the following subnet:

![Figure 3.5: Basic clock subnet](image)

The place clockset will be filled when the clock is to be set (when action $a$ is performed, correspondingly, when transition $a$ is fired), and the place clocktest will be filled when the clock expires.

When the clock is tested, for example $?c? t \rightarrow P$, then the place clocktest of the corresponding clock is linked to the transition $t$ so that $t$ can only fire only when the clock has expired. This is exactly the behaviour we would want.

A very simple example is shown below in Figure 3.6. Note that clock setting and clock testing now occurs in a separate branch in the subnet. It is therefore possible to have an arbitrary number of actions in between clock sets and clock tests.

![Figure 3.6: Simple clock setting and clock testing](image)
If a clock is reset, then we only test the most “recent” clocks. The most recent clock here refers to the closest clock to the clock test. We are not interested here in clocks in other branches, as by precluding local definitions, we know that it is not possible to test a clock which was set in another branch. Therefore, we can be certain that for each clock condition, there will be at most only one clocktest place to connect to. Figure 3.7 shows an example of this. Note that we will have to remove clocks that are left “unconnected”.

\[ P = (a < c : \exp(1.0) > \rightarrow b < c : \exp(2.0) > \rightarrow ? c ? e \rightarrow P) \]

**Figure 3.7:** Clock resetting

**Clock stopping**

For there to be a valid corresponding Petri net, clocks that can be stopped and restarted must have exponentially-distributed set times. In this case, resuming a clock is the same as resetting, from the memoryless property of the exponential distribution.

With the same idea as for basic clock setting and testing, we want to separate the clock hold and clock resume actions from the rest of the net. We therefore add two more places to the subnet created when the clock is set: clockhold and clockresume. A token is placed in these places respectively when there is a clock hold/resume action. We will also need an extra place that holds a token if the clock is being held. The clock thus has three states, running, expired, or held, and this is reflected by a token in the three places clockset, clocktest, and clockheld. Transitions are then added that changes the clock state accordingly. This subnet is shown in Figure 3.8.
The six immediate transitions correspond to the transitions that change the clock state according to the current state of the clock and the presence of tokens in \textit{clockhold} or \textit{clockresume}. To spell it out, the three possible actions when there is a token in \textit{clockhold}:

- if the clock is expired, it does nothing. A transition tests if the clock is expired. If it is, a token is removed and replaced from \textit{clocktest} (does not change the state of the clock), and a token is removed from \textit{clockhold}.

- if the clock is already held, it does nothing. Like before, a transition tests if the clock is held. If it is, it does not change the state of the clock, but a token is removed from the place \textit{clockhold}.

- if the clock is running, then the clock is held. Another transition tests if the clock is running. This time, the clock state is changed. A token is removed from \textit{clockset} and added to \textit{clockheld}. The original clock hold action is then destroyed (the token is removed from \textit{clockhold}).

There are also three analogous actions when there is a token in \textit{clockresume}:

- if clock is already expired, it does nothing.

- if clock is already running, it does nothing.

- if clock is held, a transition changes the state of the clock to running.
Local processes

So what if there are local definitions of processes? As explained before, local definitions allow for clocks to be set/ stopped/ restarted/ tested in any state in a labelled transition system. This means that it is possible for clocks to be set in different places and tested in the same place. An example is shown below where in Q we cannot determine which of the two distributions c was set with. Indeed, it is for this reason that we cannot in general replace resume actions by set actions.

\[
P = (a <c:exp(1)> \rightarrow Q \mid b <c:exp(2)> \rightarrow Q),
Q = (e <c:hold> \rightarrow f <c:resume> \rightarrow P).
\]

Note that in this example, inlining would solve the problem, but that in general is not possible as the processes may be mutually dependent (Q may refer to itself, for example).

To handle local processes, we observe that we need to know the set of possible, live clock setting subnets. Possible because clocks may possibly be set in different branches, and we are interested in all the branches. Live because clock resetting will override previous clocksets, and we are interested in only the most recent clock set. Then, we extend the rules for test, hold and resume so that they connect to each of these subnets.

The rules for hold and resume are simple: a token is placed in each of the corresponding clockhold and clockresume places. This is because extra clock hold/resume actions have no effect on the state of the clock (see previous section).

The rules for clock testing is slightly more complicated. To illustrate, consider \(?c, \, d\)?, and imagine that further that there are two clocks \(c_1\) and \(c_2\) for \(c\), and two clocks \(d_1\) and \(d_2\) for \(d\). Now, we require that both conditions \(?c\)? and \(?d\)? are satisfied before the action can be performed. This also means that either of the two clocks for \(c\) and either of the two clocks for \(d\) have expired. This is because, for the possible clock setting subnets of one clock (\(c\) for example), only one can be active, in that it has a token on one of its places. In short, if \(C_1\) is true if clock \(c_1\) has expired, then the condition \(?c, \, d\)? will be the same as the logic formula: \(((C_1 \lor C_2) \land (D_1 \lor D_2))\).

Figure 3.9 shows a subnet that satisfies this logic formula. In the figure, \(p_i^{c_1}\) is the clocktest place corresponding to clock \(c_1\), and \(t\) is the guarded transition.

3.5 Translation scheme of SFSP\(^-\) to a Petri net

In this section, a formal translation scheme for syntactically converting SFSP\(^-\) into a Petri net is presented. Most of the work in this section was done together with my supervisor and second
3.5. Translation scheme of SFSP to a Petri net

Figure 3.9: Petri net that satisfies \(((C_1 \lor C_2) \land (D_1 \lor D_2))\)

marker for a paper that was eventually not published [12].

3.5.1 Top level translation

$T_{FSP}$ is the top level translation function that takes an FSP grammar and generates a net:

$$T_{FSP}[D_1 \ldots D_n] = N_1 \otimes \ldots \otimes N_n$$

where

$$N_k = T_{D}[D_k] \quad 1 \leq k \leq n$$

$\otimes: PNet \to PNet \to PNet$ is the composition operator, where $PNet$ is the set of Petri nets. $N_1 \otimes N_2$ denotes the composition of the Petri nets $N_1$ and $N_2$. In a composition, each process has a corresponding transition for each action. If an action is shared (synchronised) between two processes, then the corresponding transitions in the two Petri nets will have to be merged. Each such transition in $N_1$ must be separately merged with every transition in $N_2$ that is labelled with the same action.

$T_D$ is a function that takes a process definition, local definitions and all, and returns the Petri net that corresponds to a process:

$$T_D[i_0 = P_0, \ldots, i_n = P_n] = \text{resolve } (\text{fix } \{N_1, \ldots, N_n\} \ \rho)\eta_n$$

where

$$\rho = \bigcup_{k=1}^{n} \left\{ \left( i_k, \bigcirc \right) \right\}$$

$$(N_0, \eta_0) = T_P[P_0] \ \rho \ \phi$$

$$(N_k, \eta_i) = T_P[P_k] \ \rho \ \eta_{k-1} \quad 1 \leq k \leq n$$

$\text{fix } : \{PNet\} \to PEnv \to PNet$ is a function that handles recursive references and combines a set of Petri nets (corresponding to local definitions of processes) into a single Petri net (the
single process definition). To handle recursive references, we pre-allocate for each process \( P \) an empty root place prior to translation of \( T_D \). Note that we are allocating for locally defined sub-processes as well (in practice, names are unique). This will be stored in the set of bindings \( \rho : PEnv \), where \( PEnv : I \rightarrow L \) and \( L \) is the set of place names. That is, \( \rho \) is a set of bindings, each mapping a process name in the FSP script to its corresponding root place. When translating a process \( P \), all recursive references to \( P \) (using the identifier \( I \)) are translated into arcs into the root place (from the environment \( \rho \)). When translation is complete, the function \( fix \) is used to overwrite the pre-allocated root place by the root of the resulting Petri net (this is always a place) and a token is added to the root place of \( P_1 \) to represent that it is the start state of the process. Note that the individual Petri nets that are translated by \( T_P \) for each call of \( T_D \) will be connected to each other, because they will refer to each other (otherwise they can be omitted), and will therefore result in a single Petri net when \( fix \) is applied.

\( resolve \) is a function that traverses the Petri nets and connects the clocks appropriately using the clock environment \( \eta : CEnv \). \( CEnv : C \rightarrow \{ (L,L,L) \} \) is a set of bindings, each mapping a clock name to a set of triples of places. The triple of places \((p_c^t, p_h^c, p_r^c)\) corresponds to the clocktest, clockhold and clockresume places of a clock subnet of clock name \( c \) that was described and explained in section 3.4.3. As explained previously, each clock name may have several clock bindings, which is why we need \( CEnv \) to return a set of triples. \( resolve \) will be discussed further in section 3.5.4.

\( \phi \) here represents an empty set of bindings.

### 3.5.2 Translating processes

\( T_P : P \rightarrow PEnv \rightarrow CEnv \rightarrow (PNet, CEnv) \) is a function that takes a process description, a process environment (of name, place bindings) and a clock environment (of name, triples of places bindings) and generates a Petri net and a clock environment. \( T_P \) must return, in addition to the subnet, a clock environment, because the process \( P \) may have several clock set actions - these must be appropriately linked up with the clock test/hold/resume actions in the other local process definitions.

In the following definitions, an arc whose source is labelled with a place name is assumed to be connected to that place. Similarly, an arc whose target is the result of subsequent translation is assumed to be connected to the root node of the net returned by the translation. In practice, all places are labelled with a unique identifier. Transitions do not need to be labelled for identity purposes; but transitions that correspond to named actions or delays in the FSP will be labelled...
3.5. Translation scheme of SFSP to a Petri net

appropriately.

\[
T_P[\text{STOP }] \rho \eta = \left( \bigcirc, \eta \right)
\]

\[
T_P[i] \rho \eta = (\rho i, \eta) \quad i \in I
\]

\[
T_P[ ( x_1 \rightarrow p_1 \mid \ldots \mid x_n \rightarrow p_n ) ] \rho \eta = (N, \bigcup_{i=1}^{n} \eta_i')
\]

where

\[
N = X_1 \ldots X_n
\]

\[
(N_i, \eta_i') = T_P[p_i] \rho \eta_i
\]

The process \text{STOP} (the first rule) is translated to a place with no out-going transition. It returns the clock environment as well, unchanged.

A reference to a process name (the second rule) is translated as a reference to the root place of the net for that process. The reference of the root place is acquired from the environment parameter \( \rho \) using the process identifier \( i \).

Choice among \( n \geq 1 \) prefixes (the third rule) is translated in two parts: first each prefix (which may be a simple action prefix, an action that sets or tests a set of clocks, or even one that may hold or resume a clock) is translated; then, each sequel is translated by recursively invoking \( T_P \). Importantly, the prefix may result in one or more clocks being (re)set. Therefore, the translation of a prefix (via \( T_X \)) must return, in addition to a sub-net, a new clock environment. An interesting point to note here is that clocks may be set in different branches of a choice, and these clocks must end up in the clock bindings. Note also that a clock set action does not reset another in a separate branch. Therefore, the clock environment returned by \( T_P \) is the union of the possibly modified clock environment (after translation of the different branches).

Note that an FSP prefix \( (x \rightarrow p) \) is the same as a single choice. For completeness, we need to add \( T_P[x \rightarrow p] = T_P[ (x \rightarrow p) ] \).
## 3.5.3 Translating prefixes

Prefixes are translated by the function $T_X : \text{Prefix} \rightarrow \text{CEnv} \rightarrow (\text{PNet}, \text{CEnv})$. Simple action prefix (e.g. $a \rightarrow P$) becomes a transition, labelled with the name of the action:

$$T_X[a] \eta = \left( \begin{array}{c} a \\ \eta \end{array} \right)$$

An action $a$ with associated clock actions becomes a transition, labelled $a$, that on firing also does the clock set/hold/resume action. This means that a token is placed in the respective place ($\text{clockset}$, $\text{clockhold}$ or $\text{clockresume}$) of the corresponding clock subnet. Translating a clock action is dealt with in the next subsection.

$$T_X[\left[ a < \text{CAct}_1, \ldots, \text{CAct}_n \right] \eta = \left( N, \bigcup_{i=1}^{n} \eta_i \right)$$

where

$$N = \bigcup_{i=1}^{n} N_i$$

$$\left( N_i, \eta_i \right) = T_{\text{CACT}}[\text{CAct}_i] \eta$$

The function for translating guarded actions is shown below:

$$T_X[\left[ ?c_1, \ldots, ?c_n \rightarrow a \right] \eta = \left( \begin{array}{c} [c_1, t], [c_n, t] \\ a \end{array} \right) \eta$$

The action is only enabled when all the clocks in the guard have expired. Therefore, the transition corresponding to the guarded action must only fire when the expiry place for every clock in the guard contains a token. Because processes may be mutually dependent, we do not in general know which clocks may be set on entry to a process until all processes have been translated. For example, process $P$ calls $Q$, $Q$ calls $R$ and $P$, and $R$ calls $P$ and $Q$. If we translate $Q$ before $R$ we may miss the fact that a clock will be set in $R$ prior to calling $Q$.

Therefore, we introduce place holders. $[c_1, t]$ is a clock condition placeholder. There are three types of place holders, clock condition placeholders ($[c_1, t]$), clock hold placeholders ($[c_1, h]$), and clock resume placeholders ($[c_1, r]$). These place holders will be replaced by the function
3.5. Translation scheme of SFSP to a Petri net

3.5.4 Translating clocks

A clock setting action sets up the clock subnet described in detail in section 3.4.3, and changes the bindings of the clock names in \( \eta \) accordingly. The individual clock setting actions are translated by \( T_{CAct} : CAct \rightarrow CEnv \rightarrow (PNet,CEnv) \)

\[
T_{CAct}[c : d] \eta = (N, \eta \ominus c + (c, \{ p^c_f, p^c_h, p^c_r \})
\]

where

\[
\ominus : CEnv \rightarrow C \rightarrow CEnv
\]

removes any binding of the form \((c, \_ )\) from \( \eta \).

Note that it may appear unnecessary to remove bindings prior to adding new ones. However, as previously explained, the clock environment parameter may contain several bindings for a clock \( c \). As clocks that were previously set higher up in a branch should be reset, all existing bindings of the clock must be removed.

Clock hold/ resume actions are very similar indeed to clock conditions. Clock hold/ resume actions involve placing a token in the clockhold or clockresume place of a clock sub-net. However, like the case in clock conditions, we do not in general know the clocks before the clock hold/ resume action. Therefore, like the clock conditions, we introduce place holders.

\[
T_{CAct}[c : hold] \eta = [c, h]
\]

\[
T_{CAct}[id : resume] \eta = [c, r]
\]
The *resolve* function replaces these place holders using the following rules:

\[
\text{replace}([c, t], \eta) = \left( \begin{array}{c}
p_1^t & \cdots & p_n^t \\
\end{array} \right)
\]

where \( \{(p_1^t, p_h^t, p_r^t) \ldots (p_n^t, p_h^t, p_r^t)\} = \eta_c \)

\[
\text{replace}([c, h], \eta) = \left( \begin{array}{c}
p_1^h & \cdots & p_n^h \\
\end{array} \right)
\]

where \( \{(p_1^h, p_1^h, p_r^h) \ldots (p_n^h, p_1^h, p_r^h)\} = \eta_c \)

\[
\text{replace}([c, r], \eta) = \left( \begin{array}{c}
p_1^r & \cdots & p_n^r \\
\end{array} \right)
\]

where \( \{(p_1^r, p_1^r, p_r^r) \ldots (p_n^r, p_1^r, p_r^r)\} = \eta_c \)

Notice how it is possible for a clock action to affect several clocks (of the same name), and how it is possible to test several clocks. In particular, note that the clock test is a *conjunction of disjunctions* (see Section 3.4.3, i.e. all clocks of different names must have expired, but only one clock of the same name needs to be expired before the guarded transition can fire.

### 3.5.5 Examples

Figure 3.10 shows an example of the translation of the process \( \mathcal{P} = (a \rightarrow \mathcal{P}) \). The figure on the left shows the result of \( \text{T}_\mathcal{P}[(a \rightarrow \mathcal{P})] \{ (\mathcal{P}, \mathcal{P}_r') \} \phi \). The reference to \( \mathcal{P} \) has been translated as an arc to the root place (labelled \( r \) in \( \rho \)). The figure on the right shows the effect of applying \( \text{fix} \).

![Figure 3.10: Simple Prefix](image)
3.5. Translation scheme of SFSP to a Petri net

Figure 3.11 shows the complete translation of the process \( \mathcal{P} = (a \rightarrow \mathcal{P} \mid b \rightarrow \mathcal{P}) \), an example of (nondeterministic) choice.

\[
P = (a \rightarrow \mathcal{P} \mid b \rightarrow \mathcal{P}).
\]

**Figure 3.11:** Choice

Figure 3.12 shows the Petri net being constructed from \( \mathcal{P} = (a <c: \exp (1.0)> \rightarrow ?c? b \rightarrow \mathcal{P}) \). The figure on the left shows the application of \( T_P \) just after the clock setting sub-net has been constructed. The figure on the right shows the complete translation. Note how the translation of \( ?c? b \rightarrow \mathcal{P} \) links to the expiry place \( p_c^t \). Note that it is possible for a clock to be (re)set, possibly many times, before it is tested. In this case a separate sub-net is built each time the clock is set and previous sub-nets are left with their expiry places unconnected. These are straightforwardly removed when the translation is complete; the details are omitted.

**Figure 3.12:** Simple clock setting and clock testing

Figure 3.13 shows an example of this. The figure on the left shows the result of applying the translation as described; the figure on the right shows the final net after removal of the redundant nodes. Note that the clock environment in the remaining call to \( T_P \) contains the latest binding for clock \( c \).

Figure 3.14 shows an example of a process where there a clock is set in different parts of the net but tested in a single sequel.
Figure 3.13: Clock resetting

\[ P = (a <c: \text{exp}(1.0)> \rightarrow b <c: \text{exp}(2.0)> \rightarrow ?c?e \rightarrow P). \]

Figure 3.14: Clock setting and testing with choice

\[ P = (a <c: \text{exp}(1.0)> \rightarrow Q \\
| b <c: \text{exp}(2.0)> \rightarrow Q), \\
Q = (?c?e \rightarrow P). \]
3.6 Alphabet extensions, hiding, relabeling and priority

3.6.1 Hiding

Hiding in SFSP is represented by $P \setminus B$, where $P$ is the process definition and $B$ is a set of action labels. The mapping of the hiding operator to the underlying stochastic automata is:

$$M(E \setminus B) = M(E) \setminus B$$

The semantics of the hiding operator on the stochastic automata can be defined as follows: Let $P = (S, s_0, A, C, \rightarrow, \kappa)$ and $B$ is a set of action labels, then

$$P \setminus B = (S, s_0, (A - B) \cup \{\tau\}, C, \rightarrow', \kappa)$$

where $\rightarrow'$ is the smallest relation satisfying

$$\frac{P \xrightarrow{a,c} P'}{P \setminus B \xrightarrow{\tau,c} P' \setminus B} a \in B \quad \frac{P \xrightarrow{a,c} P'}{P' \setminus B \xrightarrow{a,c} P' \setminus B} a \notin B$$

The translation of hiding is straightforward. The hiding is done before the composition of the processes (i.e. $M((Q_1 \parallel \ldots \parallel Q_n) \setminus B) = M(Q_1) \setminus B \parallel \ldots \parallel M(Q_n) \setminus B$). Thus any actions which are not meant to be synchronised will not be.

3.6.2 Relabeling

Relabeling in SFSP is written as $P \ / \ R$, where $P$ is the process definition and $R \subseteq L \times L$. Like hiding, the mapping of the relabeling operator is:

$$M(E / B) = M(E) / B$$

The semantics of the relabeling operator in stochastic automata is defined as follows: Let $P = (S, s_0, A, C, \rightarrow, \kappa)$ and $R \subseteq L \times L$, then

$$P / B = (S, s_0, (A - B_1) \cup B_2, C, (\rightarrow - \rightarrow_1) \cup \rightarrow_2, \kappa)$$

where

$$B_1 = \{a \in A \mid \exists a'. (a, a') \in R\}$$
\[ B_2 = \{ a' \mid \exists a \in A. (a, a') \in R \} \]

\[ \rightarrow_1 = \{ (p, a, c, p') \in \rightarrow \mid a \in B_1 \} \]

\[ \rightarrow_2 = \{ (p, a', c, p') \mid (p, a, c, p') \in \rightarrow_1 \land (a, a') \in R \} \]

Like hiding, relabling is done before the composition of processes (i.e. \( M((Q_1 \mid \ldots \mid Q_n)/R) = M(Q_1)/R \mid \ldots \mid M(Q_n)/R \)). The translation is thus straightforward.

### 3.6.3 Alphabet extensions

Alphabet extensions are represented in SFSP by \( P + B \), where \( P \) is a process definition and \( B \) is a set of action labels. The mapping of the alphabet extension operator is exactly like that shown in hiding and relabling.

Assuming \( P = (S, s_0, A, C, \rightarrow, \kappa) \) and \( B \) is a set of action labels, the alphabet extension can be defined in SA as

\[ M(E + B) = M(E) + B = (S, s_0, A \cup B, C, \rightarrow, \kappa) \]

Although alphabet extensions are easy to formalise, the translation is not so trivial. Extending the alphabet of a process that cannot do the action restricts the actions of the processes it is composed with (similar to the restrict operator in CCS and other process algebras). For example, the FSP specification \( P = (STOP + \{a\}). Q = (a->b->Q). \mid S = (P||Q) \) results in a system that cannot do anything (see fig. 3.15(a)). Following the above rules, where we “combine” transitions with the same name, we get an erroneous Petri net (see fig 3.15(b)). Instead, we must first determine the alphabets of the composed processes, check if there is an action from that process, and then do the appropriate action. If there is an action from that process, then we can compose as usual. Otherwise, like \( P \) in the example, we add a empty place to the input arc of the shared transition. This empty place prevents the transition from firing, exactly like how the alphabet extension in the example prevents \( Q \) from performing action \( a \) (see fig 3.15(c)). We can see from this example that the Petri net and the LTS behave the same way.

### 3.6.4 Priority

Priority in FSP is specified using the syntax \( P>>a \) which means that all the actions in \( P \) with the alphabet \( a \) have a lower priority than all the other actions in \( P \). Similarly, \( P<<a \) specifies that all the actions in \( P \) with the alphabet \( a \) have a higher priority than all the other actions in \( P \).
We define the mapping of priority from SFSP to SA as:

\[ M(Q << B) = M(Q) << B \quad \text{and} \quad M(Q >> B) = M(Q) >> B \]

Priority in SA is

\[
\begin{align*}
  P \xrightarrow{a,c} P' \quad & \quad \text{if } ((a \in B) \lor (\forall b \in B, P \xrightarrow{b} b)) \\
  P << B \xrightarrow{a,c} P' << B \quad & \quad \text{if } ((a \notin B) \lor (\forall b \in B, P \xrightarrow{b} b)) \\
  P \xrightarrow{a,c} P' \quad & \quad \text{if } ((a \notin B) \lor (\forall b \in B, P \xrightarrow{b} b)) \\
  P >> B \xrightarrow{a,c} P' >> B \quad & \quad \text{if } ((a \notin B) \lor (\forall b \in B, P \xrightarrow{b} b))
\end{align*}
\]

where \( P \xrightarrow{b} \) is a convenient notation for \( \not\exists c, P', P \xrightarrow{b,c} P' \), and it intuitively means that \( P \) cannot do the action \( b \).

We note that there are really only three levels of priority in FSP, high, low, and normal.

Priority in Petri nets are represented by numbers. Therefore, we can assign the transitions three levels of priority (three different numbers) corresponding to the priority levels in FSP. For example, we can set High priority as 3, Normal as 2, and Low as 1, then for each action that has a High priority, we will assign a priority of 3 to the corresponding transition and so on for Normal and Low priority actions.

Note, that immediate transitions should have higher priorities than timed transitions. As timed
transitions only belong to clocks, the priority of timed transitions cannot be specified in SFSP, and we can assign any timed transition the priority of 0.5

### 3.7 Performance measures

There are three types of performance measures in SFSP: counter, timer, and measure. Note that performance measures are not in the grammar of SFSP presented earlier. It is bolted-on to the grammar to quickly derive performance measures. The performance measures are described below:

**counter** The semantics of a counter measure is simple. It is just the number of times an action (measured by the counter) fires during a run of the simulation.

**measure** The measure measure is a system measure. It can be used to determine queue length and utilisation measures. Associated with each measure is a set of increment and decrement actions. Each increment action increments the state measure by one, while each decrement decrements the state measure by one. The time when a state change occurs is known, and that can be used to calculate the mean and variance of the state measure.

**timer** The timer measure is the most complicated of the three. It is essentially a customer oriented measure, used to determine response times. Each timer measure has a set of start and stop actions. Consider each timer measure to have a FIFO queue. Each start action places an object (say a customer) with the current time in the queue. Each stop action removes one object and retrieves the time. The time the object has stayed in the queue since the start action till the stop action can be calculated.

### 3.7.1 Translating performance measures

**Counter measures**

Counters in SFSP count the number of times an action is performed in one run of a simulation. Although this does not make sense in a Petri net, it is analogous to the rate (or throughput) of a transition.

However, a problem arises. Because all named actions in SFSP are immediate, counters specified in SFSP count only immediate actions. When the converted GSPN (Generalised Stochastic
Petri net) is analysed, all vanishing markings are removed. This means that the counters no longer have a meaning.

Therefore, counter measures in SFSP cannot be converted into a Petri net.

**Population measures**

Population measures in SFSP has an associated set of increment and decrement actions. Increment actions increment the state measure by one, while decrement actions decrement the state measure by one. Accordingly, we add a place $P$, initially empty. Each increment action (transition) will place a token in $P$, while each decrement action (transition) will remove a token from $P$. Then the mean, variance, and distribution of the number of tokens in $P$ is exactly the same as the SFSP state measure.

Note that the system in this case will become unbounded. To make the system bounded, we must limit the maximum number tokens in $P$. This can be done by introducing another place, $E$, that holds the maximum number of tokens. By inspection, we can see that the maximum number of tokens that is required is the number of increment actions. Now, each increment action (transition) will remove a token from $E$ and place it in $P$, and vice-versa for each decrement action.

The subnet of a population measure is shown below (fig. 3.16). In the figure, note that there can be more than one increment action (it can be called differently), and the number of tokens ($n$) is the number of increment actions.

![Figure 3.16: Petri net corresponding to a population measure](image)

Observe that the increment (or decrement) transitions of the population measure will synchronise with all the increment (or decrement) actions in the Petri net when it is composed.

**Timer measures**

The timer measure in SFSP the response time between a start action and a stop action, assuming a FIFO queue. It is analogous to the passage time distribution of a Petri net. However, there are
some differences.

While the response time in SFSP is the time it takes from one action to another, the passage
time in a Petri net is really the time it takes from one marking (see Section 2.4) to another. Because all transitions specified in SFSP are instantaneous, if we can determine the marking of a Petri net immediately after the firing of a the specified transition, then we can specify the passage time corresponding to the timer measure.

Another difference is that the timer measure in SFSP assumes a FIFO queue. As Ayles [2] also mentioned, such a timer is accurate only when there is a FIFO queue. Most Petri net analysers, on the other hand, only measure the time between the first time a source marking appears to the time a target marking appears. This has several implications:

- While the timer measure in SFSP correctly measures the response time for a FIFO queue, DNA-maca will only measure the service time of the server. This is because DNA-maca will “start” the clock the moment the first start action occurs (or the first time a source marking appears), and “stop” the clock the moment the first stop action (after the start) is performed. This is equivalent to the service time.

- In order to get the response time of a queuing network, it is necessary to tag customers in the queue. In simple terms, this means inserting a “special” customer into the queue. Only one tagged customer can be in a queue. Then we can measure the time between the time the tagged customer arrives to the time the tagged customer leaves. Because there will only be one tagged customer in the queue, the DNA-maca can correctly determine the result as the response time of the queue.

In general though, it is not possible to determine exactly the markings immediately after the firing of a specified transition because there may be other transitions firing elsewhere. If these transitions are immediate, then it does not make a difference (because they occur instantaneously), but if they are timed, then they will cause problems. For example, assume that after the firing of start, the place $p_1$ gets filled. We can specify the source condition as $(p_1 > 0)$. There may be several markings that satisfy this condition. Because some of these markings may occur at different times (due to timed transitions firing), the result will not be correct. In order to get around this problem, an extra place $p_e$ is added that also gets filled when start is fired, and another immediate transition $imm$ that removes a token from $p_e$. Now, if the condition is specified as $(p_1 > 0 \&\& p_e > 0)$ then there may still be several markings, but each of these markings will always happen at the same time, because no timed transition can fire before the transition $imm$. That is, the only transitions that can fire before $imm$ are immediate transitions, and these transitions occur without changing the time.
3.8 Non Markovian processes

It is important to note that as SFSP is not based on a Markovian algebra, it is the case that many processes specified by SFSP has no Markovian equivalent. The converted Petri nets are extensions of GSPNs, which can have transitions with general firing-time distributions (GEN).

For it to be a Markovian GSPN, all the timed transitions must be exponential. This is usually not the case for a general clocked process algebra.

In a restricted case, where the net has only one GEN transition enabled at any state, then there is a direct mapping to a semi-Markov process. Semi-Markov processes can be correctly evaluated with the current version of DNAmaca.

In the more general case, where the net has more than one GEN transition is enabled at any state, there is no direct mapping to a semi-Markov process (see Section 2.4.4). The Petri net model that DNAmaca uses follows the semi-Markov stochastic Petri net (SM-SPN). Recall that SM-SPNs uses the preselection rule and prd scheduling for pre-empted GEN transitions when more than one GEN transition is enabled.

Because of that, DNAmaca does not give a correct solution to the more complex issue of properly concurrently enabled GEN transitions. Therefore, although DNAmaca will give results for non-Markovian Petri nets, the results in general will not correspond with the results given by the simulation of clocked process algebras.

Another important point to note here is that clock hold and clock resume cannot work for arbitrary distributed clocks. This is because clock hold moves a token out of clockset, while clock resume moves a token back into clockset. Restarting the clock is the same as resuming a clock only in the case where the clock is exponentially distributed.
Chapter 4

Handling probability

The formalisation and translation of SFSP gets more complicated when we consider probability. Section 4.1 explains why it is much harder.

In order to formalise SFSP, we will need a new probabilistic transition system. We adapt a transition system from [27] and give a mapping from SFSP to this transition system. The formal semantics of the transition system will be presented in structured operational semantics (Section 4.2.4).

Following that, we show how to calculate the probability from the transition system, and how to construct the weights of the Petri net from these calculated probability. Some examples will also be given (Section 4.3).

4.1 New probabilistic model

The model of probability described by Ayles [2] has been changed in the current implementation of LTSA.

4.1.1 What was wrong?

In the composition of processes, Ayles asserted that probabilistic transitions occur with a higher precedence. That is, if process $P$ can do action $a$ with probability 0.5, no matter if it is shared, it will choose whether or not to do that action with probability of 0.5 before actually doing that action. It is like an invisible action, just before the action $a$, that happens with probability of 0.5. This means that if $a$ is a shared action with another process, say $Q$, and the action $a$ is
never enabled in \( Q \), then the process \( P \) will deadlock if it ever chooses to do action \( a \). It is as though the “probabilistic actions” are by themselves are not probabilistic; it is the addition of invisible probabilistic actions that makes them “probabilistic”.

With Ayles’ model of probability, the question of composing probabilistic actions is much simplified. Because probabilistic actions themselves are not probabilistic, they synchronise as though they are not probabilistic. Not only may this result in deadlock, such a model is simplistic and not representative of real-world systems.

Another criticism of this model is that of non-determinism. Although non-determinism is a useful feature in process algebras, in order to calculate performance measures, it is essential that non-determinism is absent or resolved \([27]\). Indeed, in order to construct a valid SM-SPN, each transition must have an associated weight, thus ruling out non-determinism.

### 4.1.2 What should it be?

For performance analysis, non-determinism is resolved using schedulers. In the generative approach, non-determinism is ruled out by assigning a probability to each action.

In this model, processes will not deadlock like the example described above. Let us consider the same example: \( P \) can do either \( a \) or \( b \) with probability of 0.5 each, and action \( a \) is shared with process \( Q \) that never does \( a \). \( P \) will intend to do \( a \) with a probability of 0.5, but because it is not enabled, this probability will be redistributed to its other enabled actions.

This model is slightly complicated, and it is probably best to present another example. Figure 4.1 shows the composition of a simple process:

\[
\begin{align*}
Q &= ((0.5) a | (0.5) b), \\
R &= ((0.33) a | (0.67) c), \\
| | S &= (Q | R).
\end{align*}
\]

**Figure 4.1:** Transition probabilities of a simple composition \([27]\)

To calculate the probabilities of the composition, we must consider a few scenarios:
4.2. Formalising SFSP with probability

• if both Q and R does a. That happens with probability of $\frac{1}{2} \times \frac{1}{3} = \frac{1}{6}$.

• if Q chooses to do a and R chooses to do c. That will happen with probability of $\frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$. Q will not be successful in doing action a, but R will be successful in performing action c.

• if R chooses to do a and Q chooses to do b. That happens with probability of $\frac{1}{3} \times \frac{1}{2} = \frac{1}{6}$. R will not be successful in doing action a, but Q will be successful in performing action b.

• if Q chooses to do b and R chooses to do c. That happens with probability of $\frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$.

Now, we won’t know which action occurred first, b or c. The arc between the top two lines on the right side in the figure illustrates that the combined probabilities of the two transitions add up to $\frac{1}{3}$.

Now if we assume that the scheduler is fair, that is, it assigns actions to both processes Q and R with equal probability, then we can divide that probability equally between the two shared processes. We can therefore derive the probabilities for each action. This can be generalised to any number of processes.

This model of probability is much more satisfactory that Ayle’s, but it requires more work to implement. Section 4.3 will examine how probability in SFSP can be converted into a Petri net formalism. Before that, though, we present a formalism of SFSP with this *generative* probabilistic model.

4.2 Formalising SFSP with probability

To formalise SFSP, a clocked generative probabilistic transition system (CGPTS), based on generative probabilistic transition system (GPTS) described in D’Argenio et al. [27], will be defined. We then define a mapping from SFSP to CGPTS and present the operational semantics of CGPTS in the form that most process algebras are defined.

4.2.1 Why do we need a new transition system?

Stochastic Timed Automata, the automata which SFSP was originally based (and also SPADES [39] and MoDeST [28]), do not have this model of *generative* probability. The generative probability transition system (GPTS) described in [27] do not have the expressiveness for arbitrary time delays.
Therefore, in order to formalise all the features of SFSP, it is necessary to define a new transition system. This transition system is built on the GPTS by D’Argenio et al. [27]. This transition system is described below.

4.2.2 Clocked Generative Probabilistic Transition System

CGPTS is based on GPTS, with the addition of clocks. A CGPTS is defined to be the tuple $(\Sigma, A, I, T, q)$, where

$\Sigma$ is the set of states in the CGPTS.

$A$ is the set of action labels in the CGPTS

$I$ is the set of indices. It is required to distinguish occurrences of the same probabilistic transition, and it is standard in a probabilistic setting [27, 39].

$T : \Sigma \rightarrow \text{Prob}((C \times A \times I \times \Sigma \times CA) \cup \{0\})$ is a probabilistic transition function. $\rightarrow$ represents a partial function. $C$ is a set of clock names and $CA \subseteq C \times D$ is a set of tuples of (clock, clock action). $\text{Prob}(H)$ will be elaborated later.

$q \in \Sigma$ is the starting state of the CGPTS.

A discrete probabilistic space is a structure $(\Omega, Pr)$ where

$\Omega : C \times A \times I \times \Sigma \times CA$ is a discrete sample space

$Pr$ is a probability measure on $2^\Omega$

$\text{Prob}(H)$ is the set of discrete probability spaces $(\Omega, Pr)$ such that $\Omega \subseteq H$.

Let $s \xrightarrow{g,a,i,c,p} s'$ denote $Pr_s(\{(g, a, i, s', c)\}) = p$. Intuitively, it means that at state $s$ the CGPTS can do an action $a$ (indexed by $i$) after the clocks $g$ has expired with probability $p$. After the action, it does clock actions specified by $c$ and then it goes into state $s'$.

4.2.3 Mapping SFSP to CGPTS

In this section, a complete mapping of SFSP to CGPTS is presented. Here, $Q$ ranges over process identifiers, $E$ ranges over SFSP expressions, and $M(Q)$ represents the translation of process $Q$ to a CGPTS.
### 4.2. Formalising SFSP with probability

**Process definitions**

- \( Q = E \) means that \( M(Q) = \text{def} \ M(E) \)
- \( M(\text{STOP}) = (\{s\}, \{\tau\}, \{0\}, \{\}, s) \)

**Prefix action**

Let \( M(E) = (\Sigma, A, I, T, q) \). Figure 4.2 shows a simple CGPTS corresponding to the prefix action: \( M(\text{?g?a<c>-->E}) \)

![Figure 4.2: CGPTS of \( M(\text{?g?a<c>-->E}) \)](image)

The mapping of the prefix action is given by:

\[
M(\text{?g?a<c>-->E}) = (\Sigma \cup \{p\}, A \cup \{a\}, I \cup \{u(a)\}, T', p)
\]

where

- \( p \notin \Sigma, u(a) \) is a unique index for \( a \)
- \( T'(p) = (\Omega_p, Pr_p), \forall s \in \Sigma. T'(s) = T(s) \)
- \( \Omega_p = (g, a, u(a), q, c) \)
- \( Pr(\Omega_p) = \begin{cases} 0 & \text{if g not expired} \\ 1 & \text{if g expired} \end{cases} \)

**Choice**

Let \( M(E_i) = (\Sigma_i, A_i, I_i, T_i, q_i) \), then

\[
M((w_1)\text{?g_1?a_1<c_1>-->E_1} | \ldots | M((w_n)\text{?g_n?a_n<c_n>-->E_n}) = \\
\left( \bigcup_{i=1}^{n} \Sigma_i \cup \{p\}, \bigcup_{i=1}^{n} A_i \cup \{a_1 \ldots a_n\}, \bigcup_{i=1}^{n} I_i \cup \{u(a_1) \ldots u(a_n)\}, T', p \right)
\]

where

- \( p \notin \Sigma, u(a_i) \) is a unique index for \( a_i \)
\[ T'(p) = (\Omega_p, Pr_p), \forall s \in \bigcup_{i=1}^{n} \Sigma_i, T'(s) = T(s) \]

\[ \Omega_p = \{(g_i, a_i, u(a_i), q_i, c_i) \mid 1 \leq i \leq n\} \]

\[ Pr_p(g_i, a_i, u(a_i), q_i, c_j) = \begin{cases} \frac{w_i}{\sum_{k=1}^{n} w_k} & \text{if } g \text{ expired} \\ 0 & \text{if } g \text{ not expired} \end{cases} \]

**Other mappings**

The other mapping rules are given here:

**Composition** \[ M(Q_1 \| \ldots \| Q_n) = M(Q_1) \| \ldots \| M(Q_n) \] The composition of CGPTS is dealt with in the next section.

**Alphabet extension** \[ M(E + B) = M(E) + B \]

**Recursion** The mapping rules for recursion is exactly the same as that shown in Section 3.3.1 (page 45).

**Priority** \[ M(E << B) = M(E) << B \] and \[ M(E >> B) = M(E) >> B \]

**Hiding** \[ M(E \setminus B) = M(E) \setminus B \]

**Relabling** \[ M(E / B) = M(E) / B \]

### 4.2.4 Semantics of CGPTS

Having mapped SFSP into CGPTS, we now define the semantics of CGPTS.

**Composition**

Some notation is introduced:

\[ \|_{\gamma}^{\sigma} \] is the composition between two CGPTSs. In \( P \|_{\gamma}^{\sigma} Q \), \( \sigma \) is the probability that process \( P \) will do an action first, given that the processes have already intended not to perform any shared action. Similarly, \( \gamma \) is the probability that process \( Q \) will perform the action first. Note that they do not necessarily add up to one because in general there may be many processes being composed together.

\((i, j)\) represents the indices set where \( i \) is moved while \( j \) remains the same. Similarly, \([i, j)\) is the set where \( j \) is moved while \( i \) remains the same.
S represent the set of shared actions between P and Q. If P has the action space \( A_P \) and Q has
the action space \( A_Q \) then \( S = A_P \cap A_Q \)

\( P \) stop means that the process \( P \) cannot engage in any action.

The operational semantics of composing CGPTS (adapted from [27]) is

\[
\begin{align*}
\frac{P \xrightarrow{g,a,c,p} i P' \xrightarrow{Q \text{ h,s,d,q}} j Q'}{P \parallel \sigma Q \xrightarrow{g,a,c,p,q} (i,j) P' \parallel \sigma Q} & \quad a,b \notin S \\
\frac{P \xrightarrow{g,a,c,p} i P' \xrightarrow{Q \text{ h,s,d,q}} j Q'}{P \parallel \sigma Q \xrightarrow{g,a,c,p,q} (i,j) P' \parallel \sigma Q} & \quad a \notin S \\
\frac{P \xrightarrow{g,a,c,p} i P' \xrightarrow{Q \text{ h,s,d,q}} j Q'}{P \parallel \sigma Q \xrightarrow{g,a,c,p,q} (i,0) P' \parallel \sigma Q} & \quad s \in S \\
\frac{P \xrightarrow{g,a,c,p} i P' \xrightarrow{Q \text{ h,s,d,q}} j Q'}{P \parallel \sigma Q \xrightarrow{g,a,c,p,q} (i,j) P' \parallel \sigma Q} & \quad s \in S
\end{align*}
\]

\( v(P,S) = 1 - \sum \{ p \mid P \xrightarrow{g,s,c,p} \text{ g} \in C, s \in S, c \in (C,D) \} \) is the normalisation factor. It is the probability that action \( P \) chooses to do an action that in the shared process. This normalisation is necessary to “redistribute” the probability of trying an action that cannot be performed. This normalisation is extensively discussed in [27].

**Priority**

The operational semantics for priority in CGPTS is

\[
\begin{align*}
\frac{P \xrightarrow{g,a,c,p} i P'}{P < B i P'} & \quad \text{if } ((a \in B) \lor (\forall b \in B, P \xrightarrow{b} )) \\
\frac{P \xrightarrow{g,a,c,p} i P'}{P >> B i P'} & \quad \text{if } ((a \notin B) \lor (\forall b \in B, P \xrightarrow{b} ))
\end{align*}
\]

where \( P \xrightarrow{b} \) is a convenient notation for \( \forall g,i,c,P',P \xrightarrow{g,b,c,0} i P' \), and it intuitively means that \( P \) cannot do the action \( b \).
Alphabet extension

Let $P = (\Sigma, A, I, T, q)$, then $P + B = (\Sigma, A \cup B, I, T, q)$.

Hiding

Let $P = (\Sigma, A, I, T, q)$, then

$$P \setminus B = (\Sigma, (A - B) \cup \{\tau\}, T', q)$$

where $T'$ is the smallest relation satisfying

$$\frac{P^g, a, c, p}{P \setminus B} \ x \ y \ z \ \ x' \ y' \ a \in B$$

$$\frac{P^g, a, c, p}{P \setminus B} \ x \ y \ z \ \ x' \ y' \ a \notin B$$

Relabeling

Let $P = (\Sigma, A, I, T, q)$, then

$$P / B = (\Sigma, (A - B_1) \cup B_2, I, (T - T_1) \cup T_2, q)$$

where

$$B_1 = \{a \in A \mid \exists a'.(a, a') \in R\}$$

$$B_2 = \{a' \mid \exists a \in A.(a, a') \in R\}$$

$$T_1 = \{(s, (g, a, c, p', s')) \in T \mid a \in B_1\}$$

$$T_2 = \{(s, (g, a', c, p', s')) \mid (s, (g, a, c, p', s')) \in T_1 \land (a, a') \in R\}$$

Tying up some loose ends

- What does it mean when clocks are expired? A clock $c$ is expired when $c = 0$. Note that a clock can be any arbitrary distribution. All clocks run down to 0 at the same rate. The system of time is like that of Katoen and D’Argenio [48]. In order that clocks can be held or restarted, they should have an associated boolean that indicates if they are currently held.
• How is a clock set, stopped, or restarted? Associated with each transition is a set of
tuples, \((C \times D)\), where \(C\) is a clock name and \(D\) is a clock action. When the transition
fires, the clocks \(c \in C\) is set or stopped or restarted according to the action \(d \in D\). \(d\) can
be any arbitrary distribution, or it can be a clock hold or clock resume action. If \(d\) is a
distribution, then the clock \(c\) is set to \(t\), where \(t\) comes from the distribution. If \(d\) is a
hold action, then the boolean associated with \(c\) is set to true, and the clock stops counting
down. If \(d\) is a resume action, then the boolean associated with \(c\) is set to false, and the
clock starts counting down.

• Performance measures. Performance probes should be dealt with separately, rather than
attempting to convert them to a model to be formalised. Argent-Katwala et al. [1] de-
scribes probes in more detail.

### 4.3 Translating probabilities

It is not a simple task in general to calculate the probability. As Figure 4.1 illustrated, the
probabilities can only be determined by going through all the possible interleaving of actions
from a given state, and determining of an action is shared or not. The probability of each
interleaving is the combined probability that all the actions in that interleaving was “chosen”.
The probability of that interleaving is then distributed evenly among the enabled transitions.
It sounds complicated, but the table below should help explain how the probabilities can be
determined (see Section 4.1.2 for the example).

<table>
<thead>
<tr>
<th>No.</th>
<th>P</th>
<th>Q</th>
<th>(P(P))</th>
<th>(P(Q))</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>(\frac{2}{6})</td>
<td>(\frac{1}{3})</td>
<td>(\frac{1}{6})</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>c</td>
<td>(\frac{1}{7})</td>
<td>(\frac{2}{7})</td>
<td>(\frac{3}{7})</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td>(\frac{1}{5})</td>
<td>(\frac{1}{3})</td>
<td>(\frac{1}{5})</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>c</td>
<td>(\frac{1}{2})</td>
<td>(\frac{2}{5})</td>
<td>(\frac{1}{3})</td>
</tr>
</tbody>
</table>

I will describe here, step by step, how the probabilities of actions can be calculated. First, we
reserve variables for all actions, for example \(v_a\) for action \(a\), and initialise them to 0. Next, we
go through each interleaving at a time. For interleaving 1, we see that only \(a\) can be performed.
Therefore, we add to \(v_a\) the value \(\frac{1}{6}\). In interleaving 2, only \(c\) can be performed. We add
to \(v_c\) the value of \(\frac{1}{3}\). The same argument applies to interleaving 3, and \(\frac{1}{6}\) is added to \(v_b\). In
interleaving 4, two actions, \(b\) and \(c\) are enabled. In that case, we divide the probability of that
interleaving \((\frac{1}{3})\) by the number of actions \(2\), and add that to the variables \(v_b\) and \(v_c\). At the end, the variables will have the value of the probability of the associated action.

For this simple example, there is essentially only one state where there are multiple actions enabled, and therefore we only calculate the probabilities of all the interleavings from that state. If there are more than one state, then the same procedure will have to be carried out for each state (see Example two).

After determining the probabilities of an action, we can then assign transition weights. For the simple case (with only state with multiple actions enabled), the weight of a transition is equal to the probability of the corresponding action. For the general case, the weight of a transition is the relative probability that an action is performed from a given state. The weight is not always equal to the probability that the action is performed in a state because in most situations, there will be other transitions enabled that affect the probability of firing other transitions. (Recall that the probability that a transition is fired is the weight of that transition divided by the sum of the weight of all enabled transitions - the weights need not add up to one).

Unfortunately, I cannot prove that this scheme of converting the probabilities of SFSP into a Petri net is semantically correct, but two examples below should help convince you that the behaviour of the constructed Petri net corresponds exactly to the composed LTS.

**Example one**

The SFSP script for this example is shown below:

\[
\begin{align*}
A &= ( (1) a \rightarrow \text{STOP} \mid (1) s \rightarrow \text{STOP} ). \\
B &= ( (3) s \rightarrow \text{STOP} \mid (1) c \rightarrow \text{STOP} ). \\
S &= ( A \mid \mid B ).
\end{align*}
\]

Figure 4.3 shows the composed LTS generated with the LTSA tool.

![Figure 4.3: LTS generated by the LTSA](image)
4.3. Translating probabilities

The table of interleavings and probabilities is determined and calculated from the SFSP and shown here:

<table>
<thead>
<tr>
<th>No.</th>
<th>A</th>
<th>B</th>
<th>P(A)</th>
<th>P(B)</th>
<th>Probability</th>
<th>v_a</th>
<th>v_c</th>
<th>v_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>s</td>
<td>1/2</td>
<td>3/4</td>
<td>3/8</td>
<td>5/8</td>
<td>1/8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>c</td>
<td>1/7</td>
<td>1/4</td>
<td>1/5</td>
<td>7/15</td>
<td>1/15</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>s</td>
<td>s</td>
<td>1/2</td>
<td>3/4</td>
<td>3/8</td>
<td>7/16</td>
<td>1/16</td>
<td>3/8</td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>c</td>
<td>1/2</td>
<td>1/4</td>
<td>1/5</td>
<td>2/15</td>
<td>3/15</td>
<td>3/8</td>
</tr>
</tbody>
</table>

The converted Petri net with the weight of the transition set to be equal to the probability of the corresponding action is shown in Figure 4.4. Note that the Petri net results in the same state transition diagram, with the exact same probabilities as the LTS generated by the LTSA.

![Figure 4.4: Petri net annotated with probabilities](image)

Example two

The previous examples so far only dealt with one state that had several enabled actions. If more than one state is involved, we will have to examine each of these states individually. For this example, we will be concerned with this SFSP specification:

\[
A = (a \rightarrow AA), \quad AA = ((1)b \rightarrow STOP \mid (1)c \rightarrow STOP).
\]

\[
B = (e \rightarrow BB), \quad BB = ((1)b \rightarrow STOP \mid (3)d \rightarrow STOP).
\]

\[
|\ | S = ( A \mid | B ) .
\]

The composed LTS is shown in Figure 4.5.

Note that the states with several actions enabled are states 0, 1, 5 and 7. If there is only one action enabled, then the probability that it is performed is 1. The interleavings and probabilities for the states 0, 1, 5 and 7 is shown in the table:

![Figure 4.5: Composed LTS](image)
Figure 4.5: LTS generated by the LTSA

<table>
<thead>
<tr>
<th>No.</th>
<th>A</th>
<th>B</th>
<th>P(A)</th>
<th>P(B)</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State (State 0)</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a performed (State 7)</td>
<td>1</td>
<td>e</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>b performed (State 1)</td>
<td>2</td>
<td>e</td>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>c performed (State 1)</td>
<td>3</td>
<td>e</td>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>a and e performed (State 5)</td>
<td>4</td>
<td>b</td>
<td>1</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>a and d performed (State 5)</td>
<td>5</td>
<td>d</td>
<td>1</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>a and e performed (State 5)</td>
<td>6</td>
<td>b</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
</tr>
<tr>
<td>a and d performed (State 5)</td>
<td>7</td>
<td>d</td>
<td>0.5</td>
<td>0.75</td>
<td>0.375</td>
</tr>
<tr>
<td>c and d performed (State 5)</td>
<td>8</td>
<td>b</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
</tr>
<tr>
<td>c and d performed (State 5)</td>
<td>9</td>
<td>d</td>
<td>0.5</td>
<td>0.75</td>
<td>0.375</td>
</tr>
</tbody>
</table>

The initial converted Petri net (without the annotated probabilities) is shown in Figure 4.6.

Figure 4.6: Petri net of example two, without probabilities
Let $P(x)$ be the probability of action $x$, and $w_x$ the weight of the transition corresponding to action $x$. We calculate, from State 0 in the table, that $P(a)$ is 0.5 and $P(e)$ is 0.5. Since both $w_a$ and $w_e$, has not been set, we set $w_a = 0.5$ and $w_e = 0.5$.

Then we move on to state 7. We calculate, using the same method, that $P(c)$ is 0.25 and $P(e)$ is 0.75 from that state. Now, $w_e$ has already been set. We cannot simply set $w_e = 0.75$, otherwise, the weight of $w_a$ will be wrong. We also cannot set $w_c = 0.25$ without changing $w_e$, because observe that by doing so, the probability of $c$ relative to $e$ after $a$ has fired will be different. Therefore, we set $w_c$ to be the relative probability that $c$ is performed, which is $0.25/0.75 \times 0.5 = 0.1667$. Notice that $w_c$ relative to $w_a$ is still correct, and that $w_c$ relative to $w_e$ is also correct.

We follow a similar rule for state 1, and set $w_d = 0.375/0.625 \times 0.5 = 0.3$. At state 5, we note that $P(b)$ is 0.125, $P(c)$ is 0.3125 and $P(d)$ is 0.5625. Both $w_c$ and $w_d$ is already set. So which weight ($w_c$ or $w_d$) do we use to calculate the the weight $w_b$? It turns out that using either weights will give the same answer. So $w_b = 0.125/0.3125 \times 0.1667 = 0.125/0.5625 \times 0.3 = 0.06668$.

The completed Petri nets with annotated weights is shown in Figure 4.7. By constructing the reachability graph of the Petri net, we can see that it behaves the same way as the LTS in Figure 4.5.

![Figure 4.7: Petri net of example two, with probabilities](image)
Chapter 5

Design and Implementation

This chapter will cover the design and implementation of a LTSA plug-in that implements the translation scheme. The tool can also generate results, either directly by interfacing with an external tool, or indirectly by exporting the Petri net to a another format.

There are three parts to the design and implementation:

1. Converting an SFSP specification into a Petri net data structure.
2. Exporting the Petri net into a file format that the Petri net tools can read.
3. Generating results with a click of a button and display the results in the graphical user interface of LTSA.

Several tools that have proved useful during the implementation of the plug-in are also presented at the end of this chapter.

5.1 Implementing the conversion

A translation from SFSP to a corresponding Petri net was determined. This is described in detail in Chapter 3.

It would be ideal if the data structure after parsing by LTSA can be used to construct the Petri net, so that another parser would not have to be written. The main components of LTSA is described in Section 5.1.1

Then, according to the formalism of a Petri net, a Petri net structure was determined (see Section 5.1.2).
The pseudo-code for the conversion scheme is presented in Section 5.1.3).

5.1.1 LTSA components

Figure 5.1 shows the main components of LTSA. They are described in further detail below:

**Figure 5.1:** The main LTSA components

**HPWindow** The main class of LTSA. It contains all the GUI components of the LTSA tool, all the major methods, such as \texttt{compose()} and all the IO methods, such as saving or loading from a file. As such, it is quite a large class. It contains a reference to the CompositeState.

**CompositeState** A CompositeState is a collection of CompactStates which have been chosen to be composed. It is essentially the data structure of the entire FSP specification. From the Composite state, a reference to each CompactState can be obtained and it can then be used to construct the Petri net.

**CompactState** This is a single process definition (eg. \( P = (\ldots), \ldots, \ldots \)). A CompactState consists of an array of EventStates, and an array of alphabets (Strings). Each EventState is essentially a state (with associated events) in the labelled transition system. Therefore, each EventState in the CompactState corresponds to a place in the Petri net.

**EventState** An EventState is a linked list of transitions of a CompactState object. Every transition holds the properties of the current transition and one or several pointers to other EventStates. Transitions can be probabilistic and have a pointer to one Action and one Condition.

**Action** The Action class is an abstract class with three subclasses: ClockAction, MeasureAction, and CompositeAction. Figure 5.2 illustrates the Action classes.

The ClockAction class has three subclasses: ClockHoldAction, ClockResumeAction and ClockSetAction. Associated with these actions are a clock identifier, used to identify which clock they refer to.
The Measure action has three subclasses: ActionCounterAction (corresponding to the counter measure), PopulationCounterAction (corresponding to the measure measure, and TimerAction (corresponding to the timer measure). The ActionCounterAction has one action to count. The PopulationCounterAction contains an identifier to a counter, and has two subclasses, one to increment the counter, and one to decrement the counter. The TimerAction has an identifier and two modes, one to start the timer, one to stop the timer.

Lastly, the CompositeAction is used to combine several actions into one (using the composite design pattern).

**Condition** The Condition class represents the conditions that a transition must satisfy before it is enabled. There are only two subtypes of the Condition class: ClockCondition and CompositeCondition as shown in Figure 5.3

The ClockCondition class contains a single clock identifier, used to check if the clock has expired before it is satisfied. The CompositeCondition class, like the CompositeAction class, is used to combine several conditions into one.

### 5.1.2 Petri net components

The main components of the Petri net plug-in is shown in Figure 5.4
**Figure 5.4:** Main components of the Petri net plug-in

**PetriNet** A PetriNet consists of a vector of Places and a vector of Transitions. It contains several methods to add/get/remove places or transitions to the net. A Petri net may also contain several PNMeasures.

**Place** A Place represents a place in a Petri net. It has a name and a number of tokens. Place names must be unique.

**Transition** Transition is the class representing a transition in a Petri net. It contains two vectors of places, one vector from which the arcs come, and one to which the arcs go. Transitions have a name, a priority, and weight/probability, and a DistributionType. Transitions need not have unique names.

**PetriNetConverter** This class has the responsibility for converting the CompositeState into a Petri net. It is invoked by:

```java
try {
    PetriNetConverter c = new PetriNetConverter(compositeState);
    PetriNet convertedNet = c.getPetriNet();
} catch (Exception e) {
    ...
}
```

The exceptions thrown are:

- NullCompactStateException - When there is nothing to convert.
- ClockNotSetException - When there is a clock test without a clock set.
- WarningException - for displaying warning messages.
5.1. Implementing the conversion

The main method of conversion is shown in Section 5.1.3.

**PNMeasure** A PNMeasure object has two subclasses: PNPopulation and PNTimer, corresponding to the measure and timer measures respectively\(^1\).

**DistributionType** DistributionType is an abstract class, sub-classed by 8 other distributions as shown in Figure 5.5. Note that the instantaneous transition does not have a class by itself - this is because the instantaneous transition is represented by having a null distribution type.

![Figure 5.5: DistributionTypes](image)

5.1.3 Conversion scheme

In this section, some pseudo-code for converting the CompositeState data structure in LTSA into the PetriNet data structure will be presented. Note that the code here is simplified for readability.

**convert()**

This is the main method of conversion. The top level loop goes through each CompactState in the CompositeState. Recall that each CompactState is equivalent to a single process definition in SFSP. Places are then added for each state in the CompositeState, and transitions are added for each transition in that state. Before adding the transition to the net, note that we have to synchronise on shared actions.

The other important points to note are:

- We set the place to have one token if it belongs to a starting state.
- If an action name is “tau”, we give it a unique name.

\(^1\)In 3.7.1, we discussed why the counter measure cannot be converted.
• We add an empty place to a transition if that process has that alphabet but no outgoing transition.

• We can only replace the placeholders and remove the unconnected clocks at the end of translation.

```plaintext
for each compact state cs do
  if isMeasure(cs) // each measure is a compact state by itself
    then convertMeasure(cs)
  fi
  for each state s do
    p = pn.addPlace(s) // we add a new place if it does not exist
    if isStartState(s)
      then p.setTokens(1) // add one token
    fi
    for each action a do
      if a.name == tau
        then give a a unique name
      fi
      t = createNewTransition(a)
      if !pn.contains(t)
        then pn.addTransition(t)
      else // we synchronise on each shared transition
        for each transition t2 with same name as t do
          synchronise(t, t2)
        od
      fi
    od
  od // synchronise with alphabets of a process that has no transitions
  for each alphabet a in cs do
    if !pn.contains(a)
      then pn.getTransition(a).addFrom(STOP) // add a STOP place
    fi
  od
od
```
5.1. Implementing the conversion

31 setPriority()
32 setProbability()
33 initialiseMeasures()
34 replacePlaceholders()
35 removeUnconnectedClocks()

setPriority()

The CompositeState contains two fields that specify the priorities of the actions in the CompositeState: a Vector of action names, and a boolean that indicates whether the actions that are labelled have a high or low priority. The corresponding algorithm to set the priority is straightforward:

1 for each priorityLabel lab do
2    for each transition t do
3       if t.name().equals(lab)
4          then t.setPriority(isHighPriority())
5     fi
6    od
7 od

setProbability()

Because of time constraints, the calculation of the probability of composed actions described in Section 4.3 was not implemented. Instead another method was used, and it involves extracting the calculated probabilities from the composed LTS from the LTSA. This involves getting the probability of the transition from the composed LTS and stepping through the transitions one at a time.

1 for each state s in the composition do
2    for each action a in s do
3       if state s is start state
4          then pn.setProbability(a, a.getProb()) // set probability
5     fi
6    if pn.getTrans(a).probabilitySet() // already has probability set
then continue()
fi
if ∃ a2 in s such that pn.getTrans(a2).probabilitySet()
    then p = a.getProb() / a2.getProb() * pn.getProb(a2); // weight wrt a2
    pn.setProbability(a, p)
fi
if !a.probabilitySet() // if a not set yet, set now
    then pn.setProbability(a, a.getProb()) // set probability
fi
od
od

Although this method works fine for most cases, it is not a satisfactory solution because there is no way to correspond an action name in the composite state with the transition in the Petri net if there are several actions with the same name (for example, P = ( a -> a -> P ). or “tau” actions) \(^2\). Ideally, actions in the composite state could have unique identifiers, and these can be used to identify their corresponding transitions. Unfortunately, actions and transitions can only be identified by their name, and so probabilities cannot be reliably extracted from the composed LTS.

A solution would be to implement the calculation of the probabilities of composed actions from the individual processes.

**initialiseMeasures()**

The population measure has to add two places to the net and synchronise all of its increment and decrement actions (see Section 3.7.1). The timer measure has to add a single place and a single transition for each stop and start measure and wire them up appropriately. These addition of places and transitions are artificial, in the sense that they do not correspond to the LTS of the SFSP, and so they should be carried out in a separate method. For simplicity and extensibility, the task of initialising is delegated to the measures themselves:

\(^2\)See Section 6.1.3 at page 105 for some examples of this
PNPopulation and PNTimer will add the required places and transitions and wire them up appropriately.

replacePlaceHolders()

When the net is being constructed, the PetriNetConverter maintains a list of placeholders added during the construction. When replacePlaceHolders() is called at the end of the conversion, it traverses this list, replacing one at a time. Placeholders contain enough information (pointers) so that the replacement is straightforward. Placeholders are discussed in detail in Section 3.5.3.

removeUnconnectedClocks()

This method identifies any clock subnets that are left unconnected and removes them. Although it is only important to remove clocks subnets where the clock is left untested, to simplify the Petri net and improve analysis, unnecessary hold and resume places and transitions are also removed.

In addition, this method also checks if the hold place of the clock is connected but the resume place of the clock is not. This probably means that there is an error in the specification and a warning is thrown. The net is still correct, however, and so the net is still returned.

5.2 Exporting the Petri net

5.2.1 Visitor design pattern

The Visitor design pattern[33] was used to export the Petri net data structure. Put simply, the visitor design pattern consists of two interfaces, Visitable and Visitor. To implement Visitable, a class will need to implement the method accept(Visitor v). Most of the time, the method body will be v.visit(this). To implement Visitor, a class will need to implement visit methods for Place, Transition, PetriNet, and PNMeasure.

The classes PetriNet, Place, Transition, and PNMeasures implement the Visitable interface. Note that, because PNMeasure has two subclasses PNPopulation and PNTimer, the method body for accept(Visitor v) for PNPopulation and PNTimer are different from that of
the other classes; they are: \texttt{v.visitPopulation(this)} and \texttt{v.visitTimer(this)} respectively\textsuperscript{3}.

The advantage of the visitor design pattern is that the methods for exporting into one format can be completely encapsulated in a single visitor class, instead of in each element of the data structure. This increases cohesiveness and extensibility.

The visitor classes implemented receive an OutputStream or a Writer object to output or write to. Thus it is a simple task to output to a File (by passing it a FileWriter object), a text window (by passing it a StringWriter object), or to the standard output (by passing it System.out).

Several visitor classes are implemented. XMLVisitor outputs simple PNML while the PIPEVisitor and MedusaVisitor extends the XMLVisitor and outputs PNML specifically for PIPE and for Medusa specifically. TextVisitor outputs the Petri net in plain text format for readability, and DNAmacaVisitor outputs the Petri net in DNAmaca input format. It would be a simple task to output the Petri net in different languages / formats by implementing the visitor class.

\subsection*{5.2.2 PNML}

PNML\textsuperscript{[47]} (Petri Net Markup Language) is a proposal of an XML-based standardized interchange format for Petri nets. PNML is specified with the help of RELAX NG, a schema language based on RELAX and TREX. As there are many different types of Petri nets, there are different versions of PNML.

There are currently a few tools that accept PNML as input. The Petri Net Kernel\textsuperscript{4}, PIPE\textsuperscript{5}, and Medusa\textsuperscript{[30]} are some, to name a few.

Although PIPE and Medusa both accepts PNML as input, the PNML they accept are slightly different from each other, and thus it is necessary to design two separate subclasses, PIPEVisitor and MedusaVisitor to generate PNML specific to them.

The JAXP SAX protocol was utilised in the generation of the XML. The method is as follows:

1. XMLVisitor extends XMLReader. Thus XMLVisitor can be wrapped up inside a SAXSource. This SAXSource is the input for the Transformer object, that transforms XML from an input and writes it to an output.

\textsuperscript{3}If it was \texttt{v.visit(this)}, then both \texttt{PNTimer} and \texttt{PNPopulation} would match the method declaration \texttt{visit(PNMeasure m)}.

\textsuperscript{4}http://www.informatik.hu-berlin.de/top/pnk/index.html

\textsuperscript{5}Platform Independent Petri net Editor http://petri-net.sourceforge.net/
5.2. Exporting the Petri net

2. The constructor for XMLVisitor, like all Visitors, is parameterised with a Writer or OutputStream object. Either of these two output objects are wrapped into a StreamResult. This StreamResult object is the output from the Transformer object.

3. Now, as XMLVisitor visits the Petri net, it generates appropriate SAX events. These events are then transformed by the Transformer object into the StreamResult. In this way, the Petri net is converted to XML data.

As most Petri net analysers and simulators are graphics based, PNML supports the placement (x and y positions) of places, transitions, arcs, and labels. The PetriNet data structure converted from SFSP only contains places and transitions. Labels are generated trivially (from the names of the places and transitions). Arcs can be “accumulated” during the printing of a transition, and then printed out at the end. The position of these elements are not so easily generated though. A simple graphical layout where places are laid out on one side and transitions on another was implemented. It would be interesting and useful indeed to automatically layout Petri nets, although it will not be an easy task.

5.2.3 DNAmaca

The input language of DNAmaca is described in detail in [50]. It is strictly not a Petri net specification language as it allows the specification of models which are not Petri nets. Although this may be viewed as an undesirable feature, such flexibility has proved useful in the models that it has solved so far [49].

The DNAmaca input language also allows a user to specify the solution method to use and the performance measures to solve. The passage time measure (timer) requires three inputs: the time the it should start recording, the time it should stop recording, and the interval between recordings. The user can also choose whether to get the mean, variance, standard deviation and distribution from the specified state measure (measure).

These parameters cannot be specified in SFSP, and so a graphical user interface (see Figure 5.6) was used to get these inputs from the user. These inputs will then be exported together with the net into the DNAmaca input file (will not be converted to PNML).

(TODO: should i include syntax of DNAmaca and PNML??)
5.3 Getting and displaying results

5.3.1 User Interface (plugin component)

The Petri net plugin component consists of three panels, arranged in a tabbed pane.

**Analysis** The analysis panel has a convert button that allows the user to first convert the SFSP into a Petri net format. There is also an option panel that allows the user to change or input several options specific to DNAmaca. The analysis panel has another two buttons that allows the user to analyse the net using DNAmaca, or to export the net into one of three formats (DNAmaca, PIPE and Medusa). At the bottom of the panel is an “output” text area that functions as a multiline status bar. The converted net is displayed there, the results of analysis and the current status is also printed in that output area. Note that there is also a JProgressBar to indicate that there is some process running. Figure 5.7 shows the LTSA tool with the Analysis panel selected.

**Sensitivity Analysis** The sensitivity analysis panel (see Figure 5.11) allows the user to perform sensitivity analysis on clock parameters. For example, if a user is concerned about the queue length, say, varying with the service rate, the sensitivity panel allows the user to plot a graph of queue length against the parameter. Note that it only allows one parameter to change at one time (one-way sensitivity analysis). The user can specify the smallest value, largest value, and the number of values to analyse, and the plugin will automatically run DNAmaca with the varying parameters.

**Results** The results panel displays graphs from the results generated from DNAmaca, either from the Analysis panel or from the Sensitivity panel. The graphs implementation will be discussed in Section 5.3.3. Figures 5.8 and 5.9 shows some examples of results that are
5.3. Getting and displaying results

Figure 5.7: LTSA showing the analysis panel of the plug-in

Figure 5.8: The distribution of queue length (M/M/1/15 queue, see page 111)
Figure 5.9: Passage time density (Workshop case study, where break has fixed (0.25) distribution, page 115)

Figure 5.10: Sensitivity Results

Figure 5.11: Sensitivity analysis panel
5.3. Getting and displaying results

displayed after analysis. Figure 5.10 is a screenshot of the results panel after sensitivity analysis.

5.3.2 Integrating with DNAmaca (Runtime.exec, threads)

DNAmaca is really unique in its implementation. It really consists of a collection of C++ programs that work together. Essentially, the way it carries out the analysis is as follows:

1. The DNAmaca input file is parsed and compiled into a C++ script (user.hpp)

2. The user script is then compiled (using make and g++) together with other C++ scripts to into a state space generator.

3. This state space generator is then used to create (again using make and g++) a steady state analyser, target state analyser (for passage time analysis), moments analyser (to get the moments of the passage time), performance analyser (for state measures), and passage time analyser (to determine passage time densities).

4. These small programs are then run to solve the model and determine the performance measures.

5. To generate passage time moments and densities from Semi-Markov models, it requires the use of mpirun, a standard for running parallel programs.

Because of this, it is not possible to wrap DNAmaca using JNI for use with Java classes. The only method is to use Runtime.exec(). This also means that it is very difficult to run DNAmaca from any other system.

When Runtime.exec() is called, a process is created. This process will output to its standard output and standard error, and receive input from its standard input. Because DNAmaca outputs its results and a lot of other debugging info to the standard output, it is necessary to create two separate threads to handle the output and error stream of the process.

If all the output is redirected to the text area in the plug-in, not only will it look ugly and clutter up the area, the memory it requires will be quite significant. Therefore, the stream handler thread that handles the output of the process also parses the standard output and only prints important and relevant information to the text area. At the same time, it collects the results from the stream and stores it in a data structure which can then be plotted as a graph or exported to a text file.

---

6DNAmaca outputs more than 1 Mb of data to the standard output for a typical analysis
Another issue is with DNAmaca taking a substantial amount of time to complete the analysis. If the process was run in the main thread, the user interface would seem unresponsive. Thus, a decision was taken to run the DNAmaca process in a separate thread. A cancel button was also added so that the process can be killed when the user feels like it.

### 5.3.3 Graphs

Some of the results generated from DNAmaca, for example, the response time density, should be displayed in a chart. The chart is drawn using JFreeChart\(^7\), an open source charting and graphic package for producing charts within Java programs. It has many different plot styles and allows the quick creation of charts from data. Ayles has previously used and included JFreeChart in his implementation of SFSP; thus it makes sense to make use of JFreeChart to plot graphs.

Ideally, the graph module should allow the export and import of data (perhaps to XML). The newest version of JFreeChart allows the reading of XML data into two JFreeChart data structures that can be plotted, though those two data structures do not correspond to an XY plot. In the future perhaps, JFreeChart will include an XML utility class that allows both the writing and reading of data from and to the XYDataset.

The graph module should also allow the user to zoom in at a particular section of the graph. This is not yet a feature in JFreeChart. Perhaps in future versions, JFreeChart will provide this functionality.

In view of these possible extensions, the data collected from DNAmaca can be saved into a text file. This text file can be easily parsed and any arbitrary data structure can be constructed. Thus it is possible to easily extend LTSA with a unified graphing package so that data from DNAmaca (or any other plug-in) can be drawn onto this package. In fact, if LTSA is to be extended, then it would not even be necessary to parse the text file - the data can be easily retrieved from the plug-in via a “get” method.

### 5.4 Tools used

**CVS** Concurrent Versions System is an open-source version control system. It was used to handle various revisions of files, and also served as a backup.

\(^7\)
http://www.jfree.org/jfreechart/
5.4. Tools used

**Jakarta Ant** is an open-source build system written in Java. Its most useful features include platform-independence, dependency checking and integration with JUnit and Javadoc. It simplified the building, documentation generation, unit testing and the execution of the LTSA tool.

**JUnit** is a framework for unit testing Java code. It was used extensively in the project, not only for testing purposes but also for development purposes. JUnit was used to bypass the user interface in the early development phase, and cut down a lot of time clicking buttons in the interface.

**Javadoc** is used to generate documentation from code. It is an important documentation tool and it is useful for users to understand large pieces of code. The plug-in tool was also documented with Javadoc comments during the implementation - that would be useful to future developers.
Chapter 6

Evaluation

There are two things that may go wrong. First, the translation scheme may be wrong, in that the converted Petri net does not correspond with the semantics of SFSP. Second, the conversion by the tool itself may be wrong, in that it does not generate the net it is supposed to generate according to the translation scheme. If it is proved, by formal methods, that the translation scheme is correct, then we only need to bother with whether the tool converts the net according to the scheme.

The correctness of these two issues will be examined through inspection and testing. This is carried out in two steps: first, assume that the translation scheme is correct. Then through testing the tool, we can conclude that the tool generates the Petri net according to the translation scheme. Second, we verify that the translation scheme is correct for a few examples of varying complexity by testing the Petri net with DNAmaca and other Petri net tools, and show that they generate the same behaviour.

6.1 Verification of Petri net generation

In this section, some SFSP and their corresponding Petri nets converted by the tool will be presented. Assume that for the moment, the translation scheme is correct. We show that the tool generates a Petri net that follows the translation. The Petri nets are presented in a readable text version directly generated by TextVisitor, and also in graphical form after exporting to PIPE (through PIPEVisitor).
6.1.1 Composition

A simple SFSP composition is shown below:

\[
P = (a \rightarrow a \rightarrow P).
Q = (a \rightarrow a \rightarrow Q).
||S = (P || Q).
\]

Note that here, as \(P\) and \(Q\) have 2 \(a\) actions each, there will be a total of 4 actions in the composed process. The converted net (in both text and graphical form) is shown in Figure 6.1. Observe that this corresponds exactly to the translation scheme.

![Figure 6.1: Petri net of the composed processes](image)

Composition has been extensively tested with multiple processes and shared actions. However, these will not be presented here.

6.1.2 Alphabet extensions

As an example of alphabet extensions in SFSP, consider the following:

\[
P = (a \rightarrow P) + \{b\}.
Q = (a \rightarrow b \rightarrow Q).
||S = (P || Q).
\]

The converted net is shown in Figure 6.2. Note how it corresponds with the translation scheme in Section 3.6.
6.1. Verification of Petri net generation

6.1.3 Probability

Example one

Consider Example one on page 78:

\[ A = ((1) a \rightarrow \text{STOP} \mid (1) s \rightarrow \text{STOP}). \]
\[ B = ((3) s \rightarrow \text{STOP} \mid (1) c \rightarrow \text{STOP}). \]
\[ ||S = (A||B). \]

The converted net is shown in Figure 6.3. Note that in the text version of the Petri net, the weight and type is now printed. Comparing that with Figure 4.4, we see that the tool is correct for this example.

Example two

Consider now example two (page 79): 

\[ A = (a \rightarrow AA), \quad AA = ((1)b \rightarrow \text{STOP} \mid (1)c \rightarrow \text{STOP}). \]
\[ B = (e \rightarrow BB), \quad BB = ((1)b \rightarrow \text{STOP} \mid (3)d \rightarrow \text{STOP}). \]

\[ S = (A \mid \mid B). \]

The converted net is shown in Figure 6.4.

Places:
- \( A_0 \)
- \( A_1 \)
- \( A_2 \)
- \( B_0 \)
- \( B_1 \)
- \( B_2 \)

Transitions:
- \( a \) Weight: 0.5 Type: null \( (A_0) \rightarrow (A_1) \)
- \( c \) Weight: 0.1666667 Type: null \( (A_1) \rightarrow (A_2) \)
- \( e \) Weight: 0.5 Type: null \( (B_0) \rightarrow (B_1) \)
- \( b \) Weight: 0.0666667 Type: null \( (A_1, B_1) \rightarrow (A_2, B_2) \)
- \( d \) Weight: 0.3 Type: null \( (B_1) \rightarrow (B_2) \)

Figure 6.4: Converted net from example two (see page 79)

The above examples all dealt with actions with unique names. What happens if there are many “tau” actions or several actions with the same name? Three further examples will be presented that tests this. Two of them agrees with the translation scheme; the last one shows a flaw in the implementation.

**Probabilities with hiding**

The code below is an example of a process with multiple “tau” actions.

\[ A = ((1)a \rightarrow A \mid (2)b \rightarrow A \mid (3)c \rightarrow A) \setminus \{a,b\}. \]

The converted net is shown in Figure 6.5. Notice that after the hiding, the actions \( a \) and \( b \) are combined into a single “tau”.

**Probabilities with actions of the same name**

Figure 6.6 shows the net for
6.1. Verification of Petri net generation

Places:
\[ A_0 \]

Transitions:
\[ A_0 \_\text{tau} \quad \text{Weight:}0.5 \quad \text{Type:}null \]
\[ (A_0) \rightarrow (A_0) \]
\[ c \quad \text{Weight:}0.5 \quad \text{Type:}null \]
\[ (A_0) \rightarrow (A_0) \]

Figure 6.5: Petri net with probabilities and hiding

\[ A = (a \rightarrow ((2)a\rightarrow A \mid (1)c\rightarrow A) \mid b \rightarrow A). \]

Notice that there are two \( a \) actions. The implementation handles it correctly by weighting \( c \) properly with respect to \( a \).

Places:
\[ A_0 \]
\[ A_1 \]

Transitions:
\[ a \quad \text{Weight:}0.5 \quad \text{Type:}null \]
\[ (A_0) \rightarrow (A_1) \]
\[ b \quad \text{Weight:}0.5 \quad \text{Type:}null \]
\[ (A_0) \rightarrow (A_0) \]
\[ a \quad \text{Weight:}0.5 \quad \text{Type:}null \]
\[ (A_1) \rightarrow (A_0) \]
\[ c \quad \text{Weight:}0.25 \quad \text{Type:}null \]
\[ (A_1) \rightarrow (A_0) \]

Figure 6.6: Petri net with probabilities and multiple actions with the same name

Probability breaks down ...

As discussed previously in Chapter 5, the implementation of probabilities is not satisfactory. Here is an example where the extraction of probabilities breaks down.

\[ A = ( (3)a\rightarrow A1 \mid (1)b\rightarrow A), \]
\[ A1 = ( (3)c\rightarrow A2 \mid (1)b \rightarrow A), \]
\[ A2 = ( (3)a\rightarrow \text{STOP} \mid (1)c \rightarrow \text{STOP}). \]

The converted net is shown in Figure 6.7. In this example, notice that in the process definition of \( A \), the action \( a \) is set to have a probability 3 times more than \( b \). That is, \( a \) is set to have a weight of 0.75, while \( b \) is set to have a weight of 0.25. Then in the process definition of \( A1 \), \( c \) is set to have a weight of 3 times as much as \( b \) (0.75) also. In \( A2 \), we notice that both actions \( a \) and \( c \) already have their weight set, so we do not change the weights. Notice now when the
Places:
A_0
A_1
A_2
A_3

Transitions:
\( a \) Weight: 0.75 Type: null
\((A_0) \rightarrow (A_1)\)
\( b \) Weight: 0.25 Type: null
\((A_0) \rightarrow (A_0)\)
\( c \) Weight: 0.75 Type: null
\((A_1) \rightarrow (A_0)\)
\( a \) Weight: 0.75 Type: null
\((A_1) \rightarrow (A_2)\)
\( c \) Weight: 0.75 Type: null
\((A_2) \rightarrow (A_3)\)
\( a \) Weight: 0.75 Type: null
\((A_2) \rightarrow (A_3)\)

Figure 6.7: Petri net with erroneous probability

If a token is at A_3 there is an error: a should have a weight 3 times that of c, but they have the same weight!

The error is the result of not being able to differentiate different actions with the same name. If, for example, we rename either a or c by different names, the translation would work, and the behaviour of the net will not be altered.

There are two solutions to this problem: one, to recalculate the probabilities from the process definitions; two, to uniquely identify each action (but they may still carry the same name) to facilitate the extraction of probabilities.

### 6.1.4 Clocks

**Simple clock set and test**

\[
S = ( \text{start} <c:\exp(3)> \rightarrow ?c? \text{finish} \rightarrow S )
\]

describes a simple clock set and clock test process. The graphical Petri net after exporting to PIPE is shown in Figure 6.8. The text version is omitted for clarity. The transition `clocktimeout` will be unique to each clock, but the place `condition` will be shared between clocks of the same name.
6.1. Verification of Petri net generation

Figure 6.8: Petri net of simple clock set and test in PIPE

Two clocks

The code below is an example of setting two clocks in two separate branches and testing the clock in the same sequel (refer to Figure 3.14).

\[
P = (a <c:\text{exp}(1.0)> \rightarrow Q \mid b <c:\text{exp}(2.0)> \rightarrow Q), \]
\[
Q = (?c\? e \rightarrow P). \]

The converted net is shown in Figure 6.9. The text version is omitted for clarity. Notice that it is the same as Figure 3.14.

Figure 6.9: Clock setting and testing with choice

Clock resetting

Figure 6.10 is the Petri net corresponding to

\[
P = (a <c:\text{exp}(1.0)> \rightarrow b <c:\text{exp}(2.0)> \rightarrow ?c?e \rightarrow P). \]
It is an example of clock resetting (refer to Figure 3.13). Observe that the Petri net generated corresponds to the translation scheme, in that the clock subnet generated at \( a \) is removed from the Petri net.

Places:
- \( P_0 \)
- \( P_1 \)
- \( P_2 \)
- \( P0\_CLOCKSET\_1 \)
- \( P0\_CLOCKTIMEOUT\_1 \)
- \( P\_CONDITION\_0 \)

Transitions:
- \( P0\_CLOCK\_1 \) Priority:2 Type:Exponential(2.0)
  \( (P0\_CLOCKSET\_1) \rightarrow (P0\_CLOCKTIMEOUT\_1) \)
- \( a \) Priority:5 Type:null
  \( (P_0) \rightarrow (P_1) \)
- \( b \) Priority:5 Type:null
  \( (P_1) \rightarrow (P_2, P0\_CLOCKSET\_1) \)
- \( e \) Priority:5 Type:null
  \( (P_2, P\_CONDITION\_0) \rightarrow (P_0) \)
- \( P0\_CLOCKTIMEOUT\_1 \) Priority:5 Type:null
  \( (P0\_CLOCKTIMEOUT\_1) \rightarrow (P\_CONDITION\_0) \)

**Figure 6.10:** Clock resetting

Clock hold and resume

The below shows an example where there are clock holding and clock resumes. The FSP code is

\[ P = (a<c:\text{exp}(1)> \rightarrow b<c:\text{hold}> \rightarrow c<c:\text{resume}> \rightarrow ?c?e \rightarrow P). \]

For conciseness, only the graphical net is shown in Figure 6.11.

**6.1.5 Performance measures**

**Timer measure**

An example that uses the timer measure is shown below:

\[ A = (a \rightarrow b \rightarrow A). \]

\[ \text{timer } T < a, b > \]

\[ || S = (A || T). \]
6.1. Verification of Petri net generation

Figure 6.11: Clock hold and resume

The converted net is shown in Figure 6.12. Note that this cannot be analysed because it contains no tangible markings - it is only to illustrate the construction of the timer measure. The source condition when analysing with DNAmaca is \((A_1 > 0 \& \& a_2 > 0)\) and the target condition is \((A_0 > 0 \& \& b_3 > 0)\). Note that all the markings satisfying the condition \((A_1 > 0 \& \& a_2 > 0)\) must occur at the same time, because no timed transition can fire before \textsc{T_IMMEDIATE}.

Figure 6.12: Timer measure

Population measures

The converted net corresponding to the code

\[
A = (a \rightarrow b \rightarrow A).
\]

measure \(P < a, b> \parallel S = (A \parallel P)\).

is shown in Figure 6.13. Note that here there is only one increment action, and therefore the number of tokens in \textit{P_EMPTY_1} is 1. The measure we are interested in is the mean number
110 Chapter 6. Evaluation

of tokens in $P_{FULL_1}$. The cases when there are multiple increment and decrement actions are not shown.

![Population measure](image)

**Figure 6.13:** Population measure

### 6.2 Verification of translation scheme through testing

So far, we have assumed that the translation scheme is correct. In this section, showing that the translation is correct will be attempted by verifying it in several ways where applicable. There are four methods to generate results from the SFSP code: the first two deals with the model directly; the other two deals with the generated Petri net.

1. Analyse the model by analysis and queuing theory.
2. Generate results with the simulation plug-in in LTSA.
3. Analyse the generated Petri net with DNAmaca.
4. Analyse the generated Petri net with PIPE.

Observe that the simulation will not generate exact results. The results from the simulation tool that are presented here are within a 5% confidence interval.

#### 6.2.1 Simple clock

The code below describes a simple clock set and clock test.

```plaintext
timer T < start , finish >
|| SYS = ( S || T ) .
```
Figure 6.8 shows the generated Petri net for the process $S$. Note that the addition of the timer adds places and transitions (see Figure 6.12). SYS is very similar to $S$, and so the full system is not shown.

We can observe by looking at the SFSP that the mean time from start to finish is the mean time the clock takes to expire. As the mean time of the exponentially distributed clock is the inverse of its rate, we would expect that $T$ would give the result $\frac{1}{\lambda}$.

By simulation, we do indeed get the result 0.333 for the mean clock time.

Analysis with DNAmaca also yields the correct result of 0.333 for the mean passage time from start to finish.

Analysis with PIPE is slightly more complicated, as PIPE does not automatically generate performance measures. However, through analysis with PIPE, only one tangible state in the system if found, and its mean sojourn time is 0.333.

### 6.2.2 M/M/1/K

The code for the M/M/1/K is shown in Figure 6.14. The LTS and the generated Petri net are not shown because they are too complicated.

**Analysis**

The M/M/1/K queue is one with $(K-1)$ waiting positions. In this example, $K = 6$. The M/M/1/K queue has well known solutions from queuing theory.

The probability that there are no customers in the system:

$$P_0 = \frac{1 - \rho}{1 - \rho^{K+1}}$$

The probability that there are $i$ customers in the queue:

$$P_i = P_0 \rho^i = \frac{1 - \rho}{1 - \rho^{K+1}} \rho^i$$

The mean queue length:

$$L = \sum_{i=0}^{K} iP_i$$
const N=5
float LAMBDA = 1
float MU = 2
SRVR = ( start <c:exp(MU)> -> ?c? finish -> SRVR ).

ARRIVALS = ( begin <c:exp(LAMBDA)> -> ARR ),
ARR = ( ?c? arrival <c:exp(LAMBDA)> -> ARR ).

QUEUE = Q[0],
Q[i:0..N] = ( when i < N enqueue -> Q[i+1]
                | when i > 0 start -> Q[i]
                | when i > 0 dequeue -> Q[i-1] ).

timer T <arrival, finish>
measure L <arrival, finish>
measure U <start, finish>

||SSQ = ( SRVR || QUEUE || ARRIVALS
       || T || L || U )
/ { finish / dequeue, arrival / enqueue }.

Figure 6.14: SFSP code for the Single Server Queue

And the mean response time:
\[ T = \frac{L}{(1-P_0)\mu} \]

The analysis of the above M/M/1/6 queue yields the following result:
\[ U = (1-P_0) = 0.49606 \]
\[ L = 0.93701 \]
\[ T = 0.94444 \]

Simulation

The simulation yielded the following results:
\[ U = 0.497 \pm 0.01 \]
\[ L = 0.94 \pm 0.02 \]
$T = 0.95 \pm 0.02$

**DNAmaca**

DNAmaca computed the results as:

\[
U = 0.49606 \\
L = 0.93701 \\
T = 0.5
\]

Note that although the Petri net can be analysed with PIPE, the performance measures cannot be obtained automatically. That is, one must work out the tangible markings and what they correspond to, and derive the measures manually.

From these results, we can observe that the mean queue length and utilisation of all three methods correspond with each other. The result for the timer measure $T$ is different though. This is to be expected, because as explained in Section 3.7.1 (page 65), DNAmaca will measure the service time, rather than the response time. As the service time is $\mu^{-1} = 0.5$, the result computed by DNAmaca is 0.5.

### 6.2.3 Tagged queue

In order to correctly compute response times in DNAmaca, we need to tag customers. The FSP code for a tagged queue is shown in Figure 6.15. The LTS and converted net are not shown.

The interesting part with the code is the queue. The queue can have enqueue and dequeue actions when enabled. However, once a tagenqueue action occurs (when the tagged customer comes into the queue), the queue shifts into another mode. In this mode, we keep track of the number of customers before and after the tagged customer. Enqueue will place a customer behind the tagged customer, while a dequeue will remove a customer in front of the tagged customer. When there are no more customers before the tagged customer, the queue is only allowed to do a tagdequeue. This removes the tagged customer from the queue. Note that we restrict only one tagged customer to be in the queue.

The timer then will measure the time it takes for a tagged customer to arrive at the queue till it leaves the queue. The simulation results are:

$T = 0.51 \pm 0.01$
const N = 20
float Mu = 2.0
float Lambda = 1.0
float LambdaT = 0.01

SVR = ( start <c:exp(Mu)> -> ?c? finish -> SVR ).

Arr = ( <? exp(Lambda) ?> arr -> Arr ).
TagArr = ( <? exp(LambdaT) ?> tagarr -> TagArr ).
||Arrivals = (Arr || TagArr).

Q = Q[0],
Q[i:0..N] = { when i < N enqueue -> Q[i+1]
    | when i < N tagenqueue -> TQ[i][0]
    | when i > 0 dequeue -> Q[i-1]
    },
TQ[i:0..N][j:0..N] =
    ( when (i+j+1) < N enqueue -> TQ[i][j+1]
    | when i > 0 dequeue -> TQ[i-1][j]
    | when i == 0 tagdequeue -> Q[j]
    ).
timer T <tagenqueue, tagdequeue>

||SingleServerQueue = ( SVR || Q || Arrivals || T )
    /{{dequeue,tagdequeue}/start,
      enqueue/arr, tagenqueue/tagarr }.

Figure 6.15: SFSP code for the tagged queue
6.2. Verification of translation scheme through testing

And the results from DNAmaca are:

\[ T = 0.50781 \]

6.2.4 Case study: Engineering workshop [12]

An engineering works employs \( M \) workers and one manager. At the start of each day, a batch of repair jobs are delivered to the workshop. Each job must be completed before the workers are allowed home. Each worker repeatedly retrieves a job from the input batch and carries out the necessary repair. It takes an average of \( T_1 \) hours to set up a new job (dequeue) and \( T_2 \) hours to complete the repair (done). Every so often (every \( T_3 \) hours on average), the manager signals a work break (break) at which point all workers stop what they were doing. Each break lasts on average \( T_4 \) hours, after which the workers resume (continue) the job they were doing. The number of uncompleted jobs is initially \( N \) and ends up at 0. For various distributions of the four time delays, we wish to determine the distribution of the time taken to complete all \( N \) jobs. The model in SFSP is shown in Figure 6.16.

Note that if all the time delays are non-exponential, the system can only be solved by a GSMP (simulation). If all time delays are exponential, the system can be described by a Markov process. If the work breaks are non-exponential, and if all other delays are exponential, the system can be modelled as an SMP, as only one non-exponential clock is active during the breaks. In this example, we explore the case where the work breaks are of fixed duration, and where all other delays are exponential.

A simplified version of the model is shown in Figure 6.17. Note that in the figure, only the manager and one worker is shown. The job queue and extra transitions and places are not shown.

We analyse the distribution of the passage time when the parameter \( T_4 \) (the length of the break) is varied. Figure 6.18 shows the distribution of the timer measure \( T \) for six different values for \( T_4 \). Notice that the curve becomes flatter from a break of 15 minutes to a break of 5 hours. After that, the break lengths become quite significant in changing the shape of the curve. For the 4 hour example, if a job is almost completed, but the manager signals for a break, the job completion will be pushed exactly 4 hours back. This explains the kinks in the curve.
float T1 = 0.2 // setup
float T2 = 2 // repair
float T3 = 1.5 // break every T3 hr
float T4 = 1 // lasts for 1 hr
const N = 2
const M = 2

W = (dequeue <c:exp(1/T1)> -> ?c? begin <c:exp(1/T2)> -> W1
    | noJobsLeft -> allJobsDone -> W),
W1 = (?c? done -> W
    | break <c:hold> -> continue <c:resume> -> W1).

M = (start -> M1),
M1 = (wait <c:exp(1/T3)> -> ?c? check ->
    ( break <c:fixed(T4)> -> ?c? continue -> M1
    | noJobsLeft -> allJobsDone -> M) )).

JOBS = JQ[N],
JQ[0] = (noJobsLeft -> allJobsDone -> JQ[N]),
JQ[i:1..N] = (dequeue -> JQ[i-1]).

timer T <start,allJobsDone>
||SYS = (M || [1..M]:W || [1..M]:JOBS || T) /
  ( break / [1..M].break,
    continue/[1..M].continue,
    noJobsLeft/[1..M].noJobsLeft,
    allJobsDone/[1..M].allJobsDone
  ).

**Figure 6.16:** SFSP code for the tagged queue
Figure 6.17: Composition of Manager and Worker Processes
Figure 6.18: Passage time density of the workshop example
6.3 Qualitative evaluation

6.3.1 Strengths

Easy to generate exact results

The Petri net analyser plug-in gives an extra means to the user to generate results. With a few clicks of the mouse button, the user can generate exact performance measures from the SFSP model.

By default, the SOR method is used when solving with DNAmaca, and all the performance measures will be analysed. The only input that the user may need to input is for determining the passage time distribution. In that case, DNAmaca needs to know when to start calculating from, where to end, and the intervals to calculate.

The measures are then compactly displayed in the results panel, in the form of numbers and graphs. These results can be easily compared with the results from the simulation plug-in designed by Ayles.

No extra knowledge regarding syntax is required. Any user already with the LTSA tool can make use of the plug-in.

Semi-Markov Process analysis

The use of SM-SPNs over GSPNs when converting an SFSP is straightforward. Instead of insisting that all clocks are exponential, clocks may now have arbitrary distributions. SM-SPNs allow any SFP to be converted into an SMP (although the results may not be correct). In a restricted case, when only one clocked transition is enabled at any state, then the SMP generated by the SM-SPN will be correct.

The ability to analyse semi-Markov processes is an advantage over other tools that only allow analysis with Markovian process.

Functionality

The main functionality of the plug-in, besides being able to use DNAmaca to generate results, are summarised as follows:

- Ability to export Petri net. The converted net can be exported to DNAmaca, PIPE, and Medusa. Additional formats can be implemented by extending the visitor class.
• Ability to export results. The results generated by DNAmaca can be saved to a text file for further analysis or customization of graphs and so on.

• Ability to conduct sensitivity analysis automatically. This is a trivial but useful addition that automates the process of manually tweaking the parameters of the clocks. The results are also collated and plotted. These results can also be exported.

6.3.2 Weaknesses

Portability

The tool suffers from portability issues because of its integration with DNAmaca (see Section 5.3.2). As of now, the generating of results with DNAmaca can only be done on Linux machines, with the path to DNAmaca properly set.

To get around this weakness, the ability to export the Petri net to a file was included. The Petri net can then later be run with DNAmaca.

Probability

As discussed previously, there is a flaw with extracting the probabilities from the composed process. Although this flaw never materialised during all of the verification tests, and perhaps only occur in contrived cases such as that shown in Figure 6.7, it is a flaw nonetheless.

There are two ways to solve this problem. First, in the Petri net converter, we can implement the calculation of the probability from each of the individual processes, by calculating the probability of each interleaving and by determining which actions are enabled at a state (see Section 4.3. This method, while guaranteed to work, is expensive in terms of implementation effort and conversion efficiency.

The second way to solve this problem will be to label each action in the LTS with a unique identifier. The actions may still have the same name and will still synchronise with other actions of the same name, but a separate action should have a different identifier, no matter if they have the same name or not. Extracting the probabilities will then be a simple task as the stochastic analyser in LTSA already does the calculation.
Timer measures

Timer measures in SFSP do not directly correspond to passage time measures in DNAmaca (and many other Petri net tools). This is because, as explained in Section 3.7.1, timer measures in SFSP assume a FIFO queue. While this works for most queuing systems, it is an incorrect assumption. Passage time measures in DNAmaca measure the time it takes between the moment the first source condition is satisfied till the moment the target condition is satisfied. Accordingly, DNAmaca will give the result for the service time for an ordinary timer in SFSP (see Section 6.2.2).

In order for DNAmaca to generate correct response time measures, it is necessary to tag customers. Although this can be done in SFSP, it does make the model more complicated. However, as Ayles [2] pointed out, there is a trade-off between usability and flexibility. While Ayles chose usability in defining the timer measure, DNAmaca offers flexibility.
Chapter 7

Conclusion

7.1 Discussion

7.1.1 Contributions

The project’s main contributions can be summarised as follows:

- A formal semantics of SFSP is presented by defining a mapping from SFSP to CGPTS and the operational semantics of CGPTS.

- A translation scheme to compile a clocked process algebra (SFSP) into a stochastic Petri net (SM-SPN) is presented.

- An extension to the LTSA tool that implements the scheme is developed. This extension can directly derive performance measures from DNAmaca, and can also indirectly evaluate the Petri net with external Petri net tools such as PIPE and Medusa.

- A case study showing the derivation of response time densities from a high-level model using semi-Markov process analysis is presented.

I conclude, from the course of doing this project, that:

- A feasible scheme to convert clocked process algebra, based on Stochastic Timed Automata, to a stochastic Petri net exists.

- Performance measures specified together with the process algebra are simple and adequate for rudimentary analysis.
• Although a corresponding Petri net exists for any process specified in SFSP, such a Petri net may not be analysable by existing tools, and performance measures based in SFSP do not directly translate to performance measures based on Petri nets.

• It is not difficult for system/ performance analysts to (numerically) derive performance measures from complex systems with semi-Markov behaviour.

### 7.1.2 Challenges

Below are some of the challenges faced during the implementation of the project:

• Probability. As mentioned earlier, the extraction of probabilities from the stochastic analyser is not satisfactory, as the probabilities of the actions in the composed process cannot be related to the transitions in the Petri net unless the name is unique. Two methods of solving this problem exists, and are described in Section 6.3.2

• Clocks. Translating clocks is simple and straightforward when disregarding local processes and multiple clock hold/ resume actions. It is possible to convert everything in one pass of the SFSP. However, when we consider local processes and multiple clock actions, it becomes impossible to use the one-pass conversion scheme. Placeholders have to be added that must be replaced later.

While a two-pass method seems inefficient, the second pass can reveal some important information, such as whether a clock was set before a clock test, or whether there exists a clock hold action without a clock resume action.

• Performance measures. Performance measures in SFSP do not have a direct relation to performance measures in the Petri net. Counter measures for example, do not make sense.

Timer measures in SFSP are specified in terms of actions. Timer measures in Petri nets are specified in terms of markings. There may be several markings that satisfy a single condition, and these markings may occur at different times. Thus artificial places and transitions have to be added to force these markings to occur at the same time.

• DNAmaca portability. DNAmaca (the SMP version) can only run on Linux machines, and being reliant on g++, make, and mpirun, cannot be run on any desktop. Also, it is not packaged and made available for download.
7.2 Future work

I outline below ways in which the project (or other areas) can be explored and taken further.

7.2.1 Specification of performance measures

The current method of specifying performance measures only allow the most simple measures to be extracted from the model. There have been some research into specifying performance measures, either using a logic-based approach or a process algebra based approach (see Section 2.7). An implementation of either approach to specify performance measures would be welcome.

7.2.2 Fixing probability extraction

A problem with extracting the probabilities from the LTS to the transitions exists. The problem and possible solutions to the problem is described in detail in Section 4.3 and Section 6.3.2.

7.2.3 Packaging/ Re-engineering DNAmaca

DNAmaca is an excellent tool for analysing Petri nets. However, it is not packaged and made available for use on any desktop computer. Packaging it and releasing it to the public will make it more widely known and used.

It may also be a good idea to re-engineer DNAmaca, possibly as a Java program for platform-independence. Useful features to add are:

- Java interface or API for other Java programs to make use of DNAmaca. It would then be an easy task to include DNAmaca functionality, or to create a graphical user interface for DNAmaca.

- XML input and output. XML formats are becoming the norm for data-processing programs.
7.2.4 Proof of correctness of translation scheme

In Chapter 6, several examples of varying complexity were shown to be correct. That however, does not constitute a proof. Such a proof is most likely outside the expertise of an undergraduate and most definitely outside the scope of this report.

However, it should be possible to prove its correctness. Proving the translation should be straightforward for simple processes, without probabilities, clocks, and composition. An idea to carry out this proof is as follows:

- A mapping from SFSP to CGPTS was presented in Chapter 4. Let that mapping be a function $M : SFSP \rightarrow CGPTS$.
- Let the translation of SFSP to a Petri net be $T : SFSP \rightarrow PN$
- We now need to define a function that maps a CGPTS to a Petri net or vice versa. This is the difficult part.
Appendix A

Acronyms

ACP     Algebra of Communicating Processes [8]
CCS     Calculus of Communicating Systems [55]
CSL     Continuous Stochastic Logic [4, 17]
CSP     Communicating Sequential Processes [46]
CTMC    Continuous Time Markov Chain [6]
FSP     Finite State Process [54]
GSPN    Generalised Stochastic Petri Net
IMC     Interactive Markov Chains [41]
IPC     Imperial PEPA Compiler [14]
LOTOS   Language of Temporal Ordering Specifications [29]
LTSA    Labelled Transition System Analyser [54]
MACOM   tool for Markovian Analysis of COMmunication systems [58]
MoDeST  Modelling and Description Language for Stochastic Timed systems [28]
PA      Process Algebra
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPE</td>
<td>Platform Independent Petri net Editor</td>
</tr>
<tr>
<td>PRISM</td>
<td>Probabilistic Symbolic Model Checker [53]</td>
</tr>
<tr>
<td>QPN</td>
<td>Queuing Petri Net [7]</td>
</tr>
<tr>
<td>SFSP</td>
<td>Stochastic Finite State Processes [2]</td>
</tr>
<tr>
<td>SM-SPN</td>
<td>Semi-Markov Stochastic Petri nets [17]</td>
</tr>
<tr>
<td>SPA</td>
<td>Stochastic Process Algebra</td>
</tr>
<tr>
<td>SPADES</td>
<td>Stochastic Process Algebra for Discrete Event Simulation [39]</td>
</tr>
<tr>
<td>SPN</td>
<td>Stochastic Petri Net [6]</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
</tbody>
</table>
Bibliography


